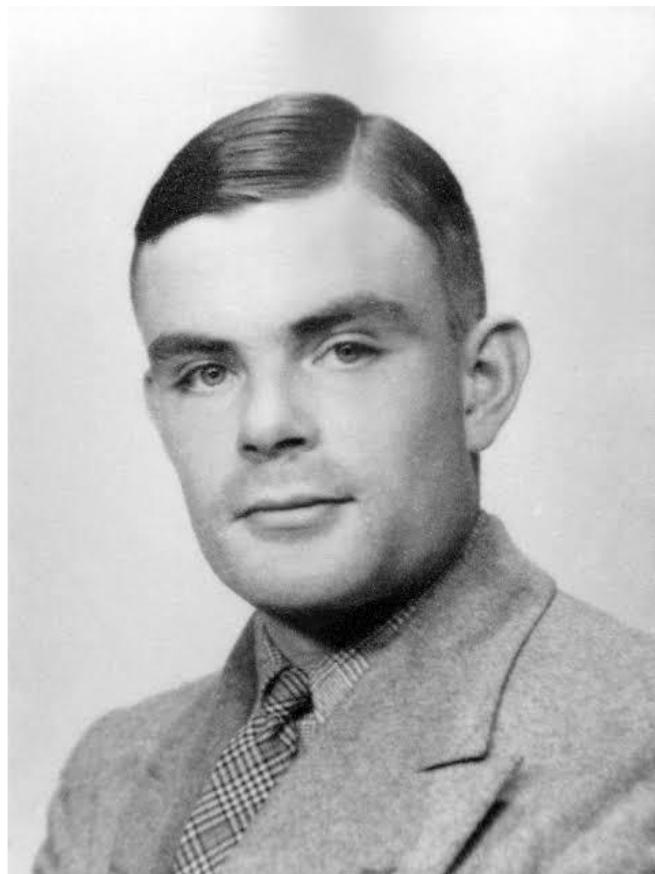


ACM Turing Award Lectures



This article presents the stimulating lectures delivered by the world's most prominent computer scientists upon their receipt of the ACM Turing Award. This collection appeal to everyone interested in the history and development of computer science, and in the perspective and thoughts, which remain relevant today.

ACM Turing Award



[Stephen Kettle's slate statue of Alan Turing at Bletchley Park](#)

Awarded for Outstanding contributions in [computer science](#)

Country [United States](#)

Presented by [Association for Computing Machinery \(ACM\)](#)

Reward(s) US \$1,000,000

First awarded 1966; 53 years ago

Website amturing.acm.org

The ACM A.M. Turing Award is an annual prize given by the [Association for Computing Machinery](#) (ACM) to an individual selected for contributions "of lasting and major technical importance to the computer field". The Turing Award is generally recognized as the highest distinction in [computer science](#) and the "[Nobel Prize of computing](#)".

The award is named after [Alan Turing](#), a British [mathematician](#) and [reader](#) in mathematics at the [University of Manchester](#). Turing is often credited as being the key founder of [theoretical computer science](#) and [artificial intelligence](#). From 2007 to 2013, the award was accompanied by an additional prize of US\$250,000, with financial support provided by [Intel](#) and [Google](#). Since 2014, the award has been accompanied by a prize of US\$1 million, with financial support provided by Google.

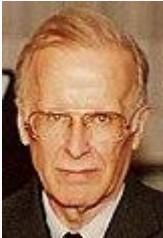
The first recipient, in 1966, was [Alan Perlis](#), of [Carnegie Mellon University](#). The first female recipient was [Frances E. Allen](#) of [IBM](#) in 2006.

Recipients

Year	Recipient	Picture	Rationale
1966	Alan J. Perlis		For his influence in the area of advanced computer programming techniques and compiler construction.
1967	Maurice Wilkes	 A black and white portrait photograph of Maurice Wilkes, an elderly man with glasses, sitting at a desk with papers and a typewriter.	Wilkes is best known as the builder and designer of the EDSAC , the first computer with an internally stored program . Built in 1949, the EDSAC used

			a mercury delay line memory . He is also known as the author, with Wheeler and Gill, of a volume on "Preparation of Programs for Electronic Digital Computers" in 1951, in which program libraries were effectively introduced.
1968	Richard Hamming		For his work on numerical methods , automatic coding systems, and error-detecting and error-correcting codes.
1969	Marvin Minsky		For his central role in creating, shaping, promoting, and advancing the field of artificial intelligence . ^[13]
1970	James H. Wilkinson		For his research in numerical analysis to facilitate the use of the high-speed digital computer, having received special recognition for his work in computations in linear algebra and "backward" error analysis.
1971	John McCarthy		McCarthy's lecture "The Present State of Research on Artificial Intelligence" is a topic that covers the area in which he has achieved considerable recognition for his work.

1972	<p><u>Edsger W. Dijkstra</u></p>		<p>Edsger Dijkstra was a principal contributor in the late 1950s to the development of the <u>ALGOL</u>, a high level <u>programming language</u> which has become a model of clarity and mathematical rigor. He is one of the principal proponents of the science and art of programming languages in general, and has greatly contributed to our understanding of their structure, representation, and implementation. His fifteen years of publications extend from theoretical articles on graph theory to basic manuals, expository texts, and philosophical contemplations in the field of programming languages.</p>
1973	<p><u>Charles W. Bachman</u></p>		<p>For his outstanding contributions to <u>database</u> technology.</p>
1974	<p><u>Donald E. Knuth</u></p>		<p>For his major contributions to the analysis of algorithms and the design of programming languages, and in particular for his contributions to "<u>The Art of Computer Programming</u>" through his well-known books in a continuous series by this title.</p>
1975	<p><u>Allen Newell</u></p>		<p>In joint scientific efforts extending over twenty</p>

	<u>Herbert A. Simon</u>		years, initially in collaboration with J. C. Shaw at the RAND Corporation , and subsequently with numerous faculty and student colleagues at Carnegie Mellon University , they have made basic contributions to artificial intelligence, the psychology of human cognition, and list processing.
1976	<u>Michael O. Rabin</u>		For their joint paper "Finite Automata and Their Decision Problem," which introduced the idea of nondeterministic machines , which has proved to be an enormously valuable concept. Their (Scott & Rabin) classic paper has been a continuous source of inspiration for subsequent work in this field.
	<u>Dana S. Scott</u>		
1977	<u>John Backus</u>		For profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on FORTRAN , and for seminal publication of formal procedures for the specification of programming languages .
1978	<u>Robert W. Floyd</u>		For having a clear influence on methodologies for the creation of efficient and reliable software, and for helping to found the following important subfields of computer science : the theory of parsing , the semantics of programming

			languages, automatic program verification , automatic program synthesis , and analysis of algorithms .
1979	<u>Kenneth E. Iverson</u>		For his pioneering effort in programming languages and mathematical notation resulting in what the computing field now knows as APL , for his contributions to the implementation of interactive systems, to educational uses of APL, and to programming language theory and practice.
1980	<u>Tony Hoare</u>		For his fundamental contributions to the definition and design of programming languages.
1981	<u>Edgar F. Codd</u>		For his fundamental and continuing contributions to the theory and practice of database management systems, esp. relational databases .
1982	<u>Stephen A. Cook</u>		For his advancement of our understanding of the complexity of computation in a significant and profound way.
1983	<u>Ken Thompson</u>		For their development of generic operating systems theory and specifically for the implementation of the UNIX operating system.

	<u>Dennis M. Ritchie</u>		
1984	<u>Niklaus Wirth</u>		For developing a sequence of innovative computer languages, <u>EULER</u> , <u>ALGOL-W</u> , <u>MODULA</u> and <u>Pascal</u> .
1985	<u>Richard M. Karp</u>		For his continuing contributions to the theory of algorithms including the development of efficient algorithms for network flow and other combinatorial optimization problems, the identification of polynomial-time computability with the intuitive notion of algorithmic efficiency, and, most notably, contributions to the theory of <u>NP-completeness</u> .
1986	<u>John Hopcroft</u>		For fundamental achievements in the design and analysis of algorithms and data structures.
	<u>Robert Tarjan</u>		

1987	John Cocke		For significant contributions in the design and theory of compilers, the architecture of large systems and the development of <u>reduced instruction set computers</u> (RISC).
1988	Ivan Sutherland		For his pioneering and visionary contributions to <u>computer graphics</u> , starting with <u>Sketchpad</u> , and continuing after.
1989	William Kahan		For his fundamental contributions to <u>numerical analysis</u> . One of the foremost experts on <u>floating-point</u> computations. Kahan has dedicated himself to "making the world safe for numerical computations."
1990	Fernando J. Corbató		For his pioneering work organizing the concepts and leading the development of the general-purpose, large-scale, <u>time-sharing</u> and resource-sharing computer systems, <u>CTSS</u> and <u>Multics</u> .
1991	Robin Milner		For three distinct and complete achievements: 1) <u>LCF</u> , the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for <u>machine assisted proof construction</u> ; 2) <u>ML</u> , the first language to include polymorphic <u>type inference</u> together with a <u>type-safe exception</u> -

			<p><u>handling</u> mechanism; 3) <u>CCS</u>, a general theory of <u>concurrency</u>. In addition, he formulated and strongly advanced <u>full abstraction</u>, the study of the relationship between <u>operational</u> and <u>denotational semantics</u>.</p>
1992	<u>Butler W. Lampson</u>		<p>For contributions to the development of distributed, personal computing environments and the technology for their implementation: <u>workstations</u>, <u>networks</u>, <u>operating systems</u>, programming systems, <u>displays</u>, <u>security</u> and <u>document publishing</u>.</p>
1993	<u>Juris Hartmanis</u>		<p>In recognition of their seminal paper which established the foundations for the field of <u>computational complexity theory</u>.</p>
	<u>Richard E. Stearns</u>		
1994	<u>Edward Feigenbaum</u>		<p>For pioneering the design and construction of large scale artificial intelligence systems, demonstrating the practical importance and potential commercial impact of artificial</p>

	<u>Raj Reddy</u>		intelligence technology.
1995	<u>Manuel Blum</u>		In recognition of his contributions to the foundations of computational complexity theory and its application to cryptography and program checking .
1996	<u>Amir Pnueli</u>		For seminal work introducing temporal logic into computing science and for outstanding contributions to program and systems verification .
1997	<u>Douglas Engelbart</u>		For an inspiring vision of the future of interactive computing and the invention of key technologies to help realize this vision.
1998	<u>Jim Gray</u>		For seminal contributions to database and transaction processing research and technical leadership in system implementation.
1999	<u>Frederick P. Brooks, Jr.</u>		For landmark contributions to computer architecture , operating systems , and software engineering .

2000	<u>Andrew Chi-Chih Yao</u>		<p>In recognition of his fundamental contributions to the <u>theory of computation</u>, including the complexity-based theory of <u>pseudorandom number generation</u>, <u>cryptography</u>, and <u>communication complexity</u>.</p>
2001	<u>Ole-Johan Dahl</u>		
	<u>Kristen Nygaard</u>		<p>For ideas fundamental to the emergence of <u>object-oriented programming</u>, through their design of the programming languages <u>Simula I</u> and <u>Simula 67</u>.</p>
2002	<u>Ronald L. Rivest</u>		
	<u>Adi Shamir</u>		<p>For <u>their ingenious contribution</u> for making <u>public-key cryptography</u> useful in practice.</p>
	<u>Leonard M. Adleman</u>		

2003	<u>Alan Kay</u>		<p>For pioneering many of the ideas at the root of contemporary <u>object-oriented programming languages</u>, leading the team that developed <u>Smalltalk</u>, and for fundamental contributions to personal computing.</p>
2004	<u>Vinton G. Cerf</u>		<p>For pioneering work on <u>internetworking</u>, including the design and implementation of the <u>Internet</u>'s basic communications protocols, <u>TCP/IP</u>, and for inspired leadership in networking.</p>
	<u>Robert E. Kahn</u>		
2005	<u>Peter Naur</u>		<p>For fundamental contributions to <u>programming language</u> design and the definition of <u>ALGOL 60</u>, to <u>compiler</u> design, and to the art and practice of computer programming.</p>
2006	<u>Frances E. Allen</u>		<p>For pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.</p>

	<u>Edmund M. Clarke</u>		
2007	<u>E. Allen Emerson</u>		For their roles in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries.
	<u>Joseph Sifakis</u>		
2008	<u>Barbara Liskov</u>		For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.
2009	<u>Charles P. Thacker</u>		For his pioneering design and realization of the Xerox Alto , the first modern personal computer, and in addition for his contributions to the Ethernet and the Tablet PC.
2010	<u>Leslie G. Valiant</u>		For transformative contributions to the theory of computation , including the theory of probably approximately correct (PAC) learning, the complexity of enumeration and of algebraic computation, and the theory of parallel and distributed computing.

2011	<u>Judea Pearl</u> ^[38]		<p>For fundamental contributions to artificial intelligence through the development of a calculus for probabilistic and causal reasoning.</p>
2012	<u>Silvio Micali</u>		<p>For transformative work that laid the complexity-theoretic foundations for the science of cryptography and in the process pioneered new methods for efficient verification of mathematical proofs in complexity theory.</p>
	<u>Shafi Goldwasser</u>		
2013	<u>Leslie Lamport</u>		<p>For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.</p>
2014	<u>Michael Stonebraker</u>		<p>For fundamental contributions to the concepts and practices underlying modern database systems.</p>
2015	<u>Martin E. Hellman</u>		<p>For fundamental contributions to modern cryptography. Diffie and Hellman's groundbreaking 1976 paper, "New Directions in</p>

	<u>Whitfield Diffie</u>		Cryptography," introduced the ideas of public-key cryptography and digital signatures, which are the foundation for most regularly-used security protocols on the Internet today.
2016	<u>Tim Berners-Lee</u>		For inventing the <u>World Wide Web</u> , the first <u>web browser</u> , and the fundamental protocols and algorithms allowing the Web to scale.
2017	<u>John L. Hennessy</u>		For pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.
	<u>David A. Patterson</u>		
2018	<u>Yoshua Bengio</u>		For conceptual and engineering breakthroughs that have made <u>deep neural networks</u> a critical component of computing.
	<u>Geoffrey Hinton</u>		
	<u>Yann LeCun</u>		

A. M. Turing Award Lecture:
Verification Engineering:
A Future Profession

Amir Pnueli

Weizmann Institute of Science

It is time that formal verification (of both software and hardware systems) be demoted from an art practiced by the enlightened few to an activity routinely and mundanely performed by a cadre of Verification Engineers (a new profession), as a standard part of the system development process.

In the talk, we will assess how far we are from the realization of this obvious necessity in terms of:

1. The current state of the art of existing verification approaches and tools of the two prevalent kinds: Model Checking and Deductive Verification. The power, accessibility, friendliness, ease of use and operation of existing tools, and the size of systems they can verify. The qualification and sophistication required of a potential user.
2. The degree of acceptability and future expectations of formal verification within different industrial sectors.
3. The educational background and a proposed curriculum for the new discipline of Verification Engineering.
4. Integration of formal verification with other

commonly practiced methods of validation such as testing and simulation.

5. The main obstacles standing in the way: the needs for user interaction. We will analyze the various places where user creativity is called for, such as in the design of powerful abstractions, reduction by symmetry and similarity, and the construction of auxiliary invariants and progress measures. It will be shown that, for particular classes of applications, it is possible to identify verification patterns and application-specific heuristics that can significantly reduce the amount of fresh intellectual effort needed for the consideration of each new instance of a system in this class.

Re-usability of verification modules should go hand in hand with re-usability of system modules.

What Next?

A Dozen Information-Technology Research Goals

Jim Gray

June 1999

Technical Report

MS-TR-99-50

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

What Next?

A Dozen Information-Technology Research Goals¹

Jim Gray

Microsoft Research

301 Howard St. SF, CA 94105, USA

Abstract: Charles Babbage's vision of computing has largely been realized. We are on the verge of realizing Vannevar Bush's Memex. But, we are some distance from passing the Turing Test. These three visions and their associated problems have provided long-range research goals for many of us. For example, the scalability problem has motivated me for several decades. This talk defines a set of fundamental research problems that broaden the Babbage, Bush, and Turing visions. They extend Babbage's computational goal to include highly-secure, highly-available, self-programming, self-managing, and self-replicating systems. They extend Bush's Memex vision to include a system that automatically organizes, indexes, digests, evaluates, and summarizes information (as well as a human might). Another group of problems extends Turing's vision of intelligent machines to include prosthetic vision, speech, hearing, and other senses. Each problem is simply stated and each is orthogonal from the others, though they share some common core technologies

1. Introduction

This talk first argues that long-range research has societal benefits, both in creating new ideas and in training people who can make even better ideas and who can turn those ideas into products. The education component is why much of the research should be done in a university setting. This argues for government support of long-term university research. The second part of the talk outlines sample long-term information systems research goals.

I want to begin by thanking the ACM Awards committee for selecting me as the 1998 ACM Turing Award winner. Thanks also to Lucent Technologies for the generous prize.

Most of all, I want to thank my mentors and colleagues. Over the last 40 years, I have learned from many brilliant people. Everything I have done over that time has been a team effort. When I think of any project, it was Mike and Jim, or Don and Jim, or Franco and Jim, or Irv and Jim, or Andrea and Jim, or Andreas and Jim, Dina and Jim, or Tom and Jim, or Robert and Jim, and so on to the present day. In every case it is hard for me to point to anything that I personally did: everything has been a collaborative effort.. It has been a joy to work with these people who are among my closest friends.

More broadly, there has been a large community working on the problems of making automatic and reliable data stores and transaction processing systems. I am proud to have been part of this effort, and I am proud to be chosen to represent the entire community. Thank you all!

¹ The Association of Computing Machinery selected me as the 1998 A.M. Turing Award recipient. This is approximately the text of the talk I gave in receipt of that award. The slides for that talk are at <http://research.microsoft.com/~Gray/Talks/Turing2.ppt>

1.1. Exponential Growth Means Constant Radical Change.

Exponential growth has been driving the information industry for the last 100 years. Moore's law predicts a doubling every 18 months. This means that in the next 18 months there will be as much new storage as all storage ever built, as much new processing as all the processors ever built. The area under the curve in the next 18 months equals the area under the curve for all human history.

In 1995, George Glider predicted that deployed bandwidth would triple every year, meaning that it doubles every 8 months. So far his prediction has been pessimistic: deployed bandwidth seems to be growing faster than that!

This doubling is only true for the underlying technology, the scientific output of our field is doubling much more slowly. The literature grows at about 15%, per year, doubling every five years.

Exponential growth cannot go on forever. E. coli (bacteria in your stomach) double every 20 minutes. Eventually something happens to limit growth. But, for the last 100 years, the information industry has managed to sustain this doubling by inventing its way around each successive barrier. Indeed, progress seems to be accelerating (see Figure 1). Some argue that this acceleration will continue, while others argue that it may stop soon – certainly if we stop innovating it will stop tomorrow.

These rapid technology doublings mean that information technology must constantly redefine itself: many things that were impossibly hard ten years ago, are now relatively easy. Tradeoffs are different now, and they will be very different in ten years.

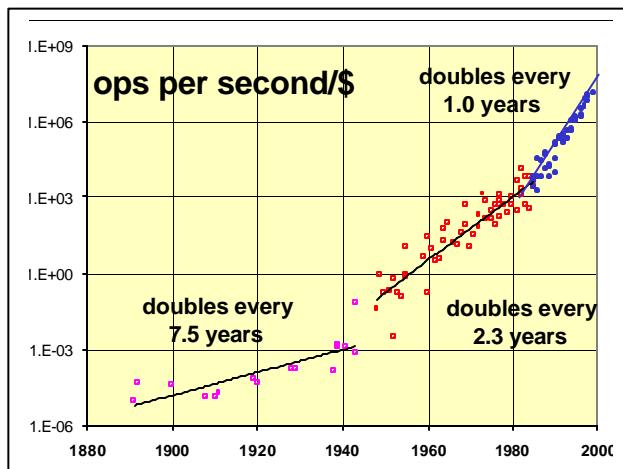


Figure 1: Graph plots performance/price versus time, where
 $\frac{\text{Performance}}{\text{Price}} = \frac{(\text{operations-per-second})}{(\text{bits-per-op})}$
Performance = (operations-per-second) x (bits-per-op)
Price = system price for 3 years
Performance/price improvements seem to be accelerating.
There appear to be three growth curves: (1) Before
transistors (1890-1960), performance/price was doubling
every seven years. (2) With discrete electronics
performance/price was doubling every 2.3 years between
1955 and 1985. (3) Since 1985 performance/price has
doubled every year with VLSI. Sources Hans Moravec,
Larry Roberts, and Gordon Bell [1].

1.3. Cyberspace is a New World

One way to think of the Information Technology revolution is to think of cyberspace as a new continent -- equivalent to discovery of the Americas 500 years ago. Cyberspace is transforming the old world with new goods and services. It is changing the way we learn, work, and play. It is already a trillion dollar per year industry that has created a trillion dollars of wealth since 1993. Economists believe that 30% of the United States economic growth comes from the IT industry. These are high-paying high-export industries that are credited with the long boom – the US economy has skipped two recessions since this boom started.

With all this money sloshing about, there is a gold rush mentality to stake out territory. There are startups staking claims, and there is great optimism. Overall, this is a very good thing.

1.4. This new world needs explorers, pioneers, and settlers

Some have lost sight of the fact that most of the cyberspace territory we are now exploiting was first explored by IT pioneers a few decades ago. Those prototypes are now transforming into products.

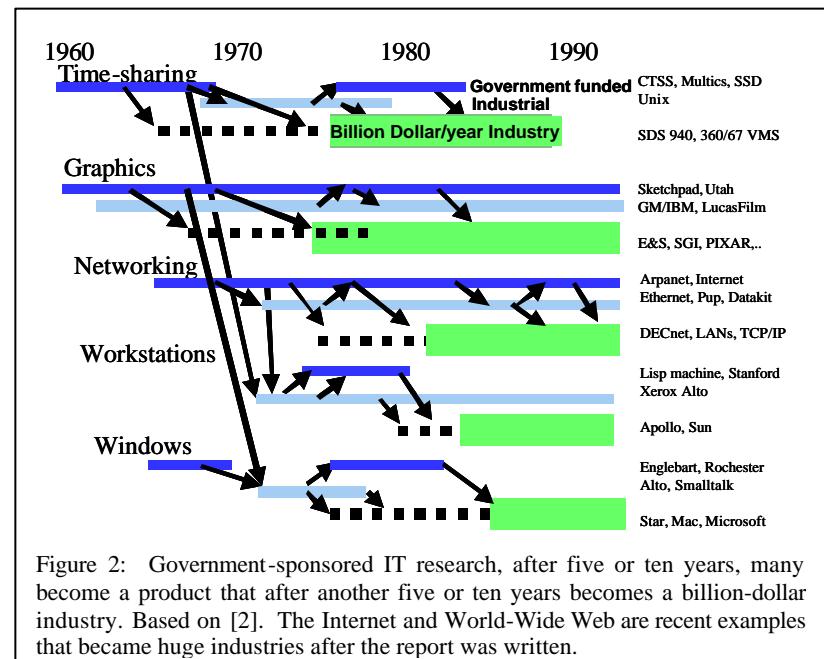
The gold rush mentality is casing many research scientists to work on near-term projects that might make them rich, rather than taking a longer term view. Where will the next generation get its prototypes if all the explorers go to startups? Where will the next generation of students come from if the faculty leave the universities for industry?

Many believe that it is time to start Lewis and Clark style expeditions into cyberspace: major university research efforts to explore far-out ideas, and to train the next generation of research scientists. Recall that when Tomas Jefferson bought the Louisiana Territories from France, he was ridiculed for his folly. At the time, Jefferson predicted that the territories would be settled by the year 2000. To accelerate this, he sent out the Lewis & Clark expedition to explore the territories. That expedition came back with maps, sociological studies, and a corps of explorers who led the migratory wave west of the Mississippi [6].

We have a similar opportunity today: we can invest in such expeditions to create the intellectual and human seed corn for the IT industry of 2020. It is the responsibility of government, industry, and private philanthropy to make this investment for our children, much as our parents made this investment for us.

1.5. Pioneering research pays off in the long-term

To see what I mean, I recommend you read the NRC Brooks Southerland report, *Evolving the High-Performance Computing and Communications Initiative to Support the nations Information Infrastructure* [2] or more recently: *Funding the Revolution* [3]. Figure 2 is based on a figure that appears in both reports. It shows how government-sponsored and industry-sponsored research in Time Sharing turned into a billion dollar industry after a decade. Similar things happened with research on graphics, networking, user interfaces, and many other fields. Incidentally, much of this research work fits within Pasteur's Quadrant [5], IT



research generally focuses on fundamental issues, but the results have had enormous impact on and benefit to society.

Closer to my own discipline, there was nearly a decade of research on relational databases before they became products. Many of these products needed much more research before they could deliver on their usability, reliability, and performance promises. Indeed, active research on each of these problems continues to this day, and new research ideas are constantly feeding into the products. In the mean time, researchers went on to explore parallel and distributed database systems that can search huge databases, and to explore data mining techniques that can quickly summarize data and find interesting patterns, trends, or anomalies in the data. These research ideas are just now creating another billion-dollar-per-year industry.

Research ideas typically need a ten year gestation period to get to products. This time lag is shortening because of the gold rush. Research ideas still need time to mature and develop before they become products.

1.6. Long-term research is a public good

If all these billions are being made, why should government subsidize the research for a trillion-dollar a year industry? After all, these companies are rich and growing fast, why don't they do their own research?

The answer is: "most of them do." The leading IT companies (IBM, Intel, Lucent, Hewlett Packard, Microsoft, Sun, Cisco, AOL, Amazon,...) spend between 5% and 15% of their revenues on Research and Development. About 10% of that R&D is not product development. I guess about 10% of that (1% of the total) is pure long-term research not connected to any near term product (most of the R of R&D is actually advanced development, trying to improve existing products). So, I guess the IT industry spends more than 500 million dollars on long-range research, which funds about 2,500 researchers. This is a conservative estimate, others estimate the number is two or three times as large. By this conservative measure, the scale of long-term industrial IT research is comparable to the number of tenure-track faculty in American computer science departments.

Most of the IT industry does fund long-range IT research; but, to be competitive some companies cannot. MCI-WorldCom has no R&D line item in the annual report, nor does the consulting company EDS. Dell computer has a small R&D budget. In general, service companies and systems integrators have very small R&D budgets.

One reason for this is that long-term research is a social good, not necessarily a benefit to the company. AT&T invented the transistor, UNIX, and the C and C++ languages. Xerox invented Ethernet, bitmap printing, iconic interfaces, and WYSIWYG editing. Other companies like Intel, Sun, 3Com, HP, Apple, and Microsoft got the main commercial benefits from this research. Society got much better products and services -- that is why the research is a public good.

Since long-term research is a public good, it needs to be funded as such: making all the boats rise together. That is why funding should come in part from society: industry is paying a tax by doing long-term research; but, the benefits are so great that society may want to add to that, and fund university research. Funding university research has the added benefit of training the next

generations of researchers and IT workers. I do not advocate Government funding of industrial research labs or government labs without a strong teaching component.

One might argue that US Government funding of long-term research benefits everyone in the world. So why should the US fund long-term research? After all, it is a social good and the US is less than 10% of the world. If the research will help the Europeans and Asians and Africans, the UN should fund long-term research.

The argument here is either altruistic or jingoistic. The altruistic argument is that long-term research is an investment for future generations world-wide. The jingoistic argument is that the US leads the IT industry. US industry is extremely good at transforming research ideas into products – much better than any other nation.

To maintain IT leadership, the US needs people (the students from the universities), and it needs new ideas to commercialize. But, to be clear, this is a highly competitive business, cyberspace is global, and the workers are international. If the United States becomes complacent, IT leadership will move to other nations.

1.7. The PITAC report and its recommendations.

Most of my views on this topic grow out of a two year study by the Presidential IT Advisory Committee (PITAC) <http://www.ccic.gov/ac/report/> [4]. That report recommends that the government sponsor Lewis and Clark style expeditions to the 21st century, it recommends that the government double university IT research funding – and that the funding agencies shift the focus to long-term research. By making larger and longer-term grants, we hope that university researchers will be able to attack larger and more ambitious problems.

It also recommends that we fix the near-term staff problem by facilitating immigration of technical experts. Congress acted on that recommendation last year: adding 115,000 extra H1 visas for technical experts. The entire quota was exhausted in 6 months: the last FY99 H1 visas were granted in early June.

2. Long Range IT Systems Research Goals

Having made a plea for funding long-term research. What exactly are we talking about? What are examples of long-term research goals that we have in mind? I present a dozen examples of long-term systems research projects. Other Turing lectures have presented research agendas in theoretical computer science. My list complements those others.

2.1. What Makes a Good Long Range Research Goal?

Before presenting my list, it is important to describe the attributes of a good goal. A good long-range goal should have five key properties:

Understandable: The goal should be simple to state. A sentence, or at most a paragraph should suffice to explain the goal to intelligent people. Having a clear statement helps recruit colleagues and support. It is also great to be able to tell your friends and family what you actually do.

Challenging: It should not be obvious how to achieve the goal. Indeed, often the goal has been around for a long time. Most of the goals I am going to describe have been explicit or implicit goals for many years. Often, there is a camp who believe the goal is impossible.

Useful: If the goal is achieved, the resulting system should be clearly useful to many people -- I do not mean just computer scientists, I mean people at large.

Testable: Solutions to the goal should have a simple test so that one can measure progress and one can tell when the goal is achieved.

Incremental: It is very desirable that the goal has intermediate milestones so that progress can be measured along the way. These small steps are what keep the researchers going.

2.2. Scalability: a sample goal

To give a specific example, much of my work was motivated by the *scalability* goal described to me by John Cocke. The goal is to devise a software and hardware architecture that scales up without limits. Now, there has to be some kind of limit: billions of dollars, or giga-watts, or just space. So, the more realistic goal is to be able to scale from one node to a million nodes all working on the same problem.

1. **Scalability:** Devise a software and hardware architecture that scales up by a factor for 10^6 . That is, an application's storage and processing capacity can automatically grow by a factor of a million, doing jobs faster (10^6 x speedup) or doing 10^6 larger jobs in the same time (10^6 x scaleup), just by adding more resources.

Attacking the scalability problem leads to work on all aspects of large computer systems. The system grows by adding modules, each module performing a small part of the overall task. As the system grows, data and computation has to migrate to the new modules. When a module fails, the other modules must mask this failure and continue offering services. Automatic management, fault-tolerance, and load-distribution are still challenging problems.

The benefit of this vision is that it suggests problems and a plan of attack. One can start by working on automatic parallelism and load balancing. Then work on fault tolerance or automatic management. One can start by working on the 10x scaleup problem with an eye to the larger problems.

My particular research focused on building highly-parallel database systems, able to service thousands of transactions per second. We developed a simple model that describes when transactions can be run in parallel, and also showed how to automatically provide this parallelism. This work led to studies of why computers fail, and how to improve computer availability. Lately, I have been exploring very large database applications like <http://terraserver.microsoft.com/> and <http://www.sdss.org/>.

Returning to the scalability goal, how has work on scalability succeeded over the years? Progress has been astonishing, for two reasons.

1. There has been a lot of it.
2. Much of it has come from an unexpected direction – the Internet.

The Internet is a world-scale computer system that surprised us all. A computer system of 100 million nodes, and now merely doubling in size each year. It will probably grow to be much larger. The PITAC worries that we do not know how to scale the network and the servers. I share that apprehension, and think that much more research is needed on protocols and network engineering.

On the other hand, we do know how to build huge servers. Companies have demonstrated single systems that can process a billion transactions per day. That is comparable to all the cash transactions in the US in a day. It is comparable to all the AOL interactions in a day. It is a lot.

In addition, these systems can process a transaction for about a micro-dollar. That is, they are very cheap. It is these cheap transactions that allow free access to the Internet data servers. In essence, accesses can be paid for by advertising.

Through a combination of hardware (60% improvement per year) and software (40% improvement per year) performance and price performance have doubled every year since 1985. Several more years of this progress are in sight (see Figures 1 and 3.)

Still, we have some very dirty laundry. Computer scientists have yet to make parallel programming easy. Most of the scalable systems like databases, file servers, and online transaction processing are embarrassingly parallel. The parallelism comes from the application. We have merely learned how to preserve it, rather

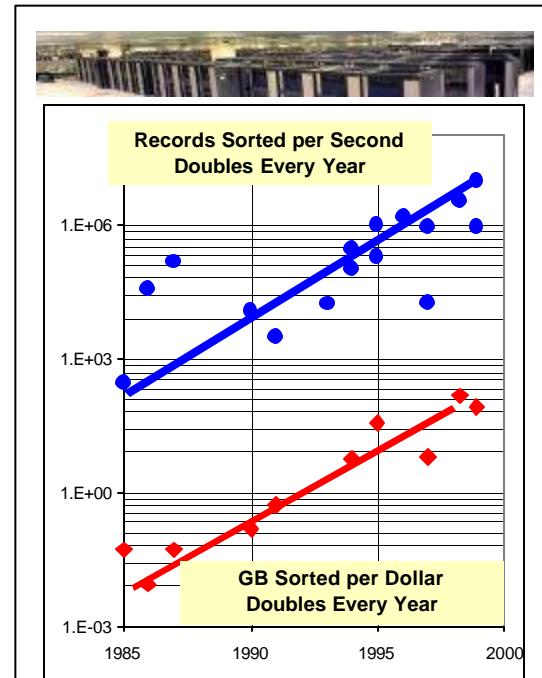


Figure 3: Top: a picture of the 6,000 node computer at LANL. Bottom: Scalability of computer systems on the simple task of sorting data. Sort speed and sorting price-performance has doubled each year over the last 15 years. This is progress is partly due to hardware and partly due to software.

than creating it automatically.

When it comes to running a big monolithic job on a highly parallel computer, there has been modest progress. Parallel database systems that automatically provide parallelism and give answers more quickly, have been very successful. Parallel programming systems that ask the programmer to explicitly write parallel programs have been embraced only as a last resort. The best examples of this are the Beowulf clusters used by scientists to get very inexpensive supercomputers (<http://www.beowulf.org/>), and the huge ASCI machines, consisting of thousands of processors (see top of Figure 3.). Both these groups report spectacular performance, but they also report considerable pain.

Managing these huge clusters is also a serious problem. Only some of the automation postulated for scaleable systems has been achieved. Virtually all the large clusters have a custom-built management system. We will return to this issue later.

The scalability problem will become more urgent in the next decade. It appears that new computer architectures will have multiple execution streams on a single chip: so each processor chip will be an SMP (symmetric multi-processor). Others are pursuing processors imbedded in memories, disks, and network interface cards (e.g. <http://iram.cs.berkeley.edu/istore/>). Still another trend is the movement of processors to Micro-Electro-Mechanical Systems (MEMS). Each 10\$ MEMS will have sensors, effectors, and onboard processing. Programming a collection of a million MEMS systems is a challenge [7].

So, the scaleability problem is still an interesting long-term goal. But, in this lecture, I would like to describe a broad spectrum of systems-research goals.

2. Long-term IT Systems Research Goals

In looking for the remaining eleven long-term research problems I read the previous Turing lectures, consulted many people, and ultimately settled on organizing the problems in the context of three seminal visionaries of our field. In the 1870s Charles Babbage had the vision of programmable computers that could store information and could compute much faster than people. In the 1940's Vannevar Bush articulated his vision of a machine that stored all human knowledge. In 1950, Alan Turing argued that machines would eventually be intelligent.

The problems I selected are systems problems. Previous Turing talks have done an excellent job of articulating an IT theory research agenda. Some of the problems here necessarily have an "and prove it" clause. These problems pose challenging theoretical issues. In picking the problems, I tried to avoid specific applications – trying rather to focus on the core issues of information technology that seem generic to all applications.

One area where I wish I had more to say, is the topic of ubiquitous computing. Alan Newell first articulated the vision of an intelligent universe in which every part of our environment is intelligent and networked [8]. Many of the research problems mentioned here bear on this ubiquitous computing vision, but I have been unable to crisply state a specific long-term research goal that is unique to it.

3. Turing's vision of machine intelligence

To begin, recall Alan Turing's famous "Computing Machinery and Intelligence" paper published in 1950 [9]. Turing argued that in 50 years, computers would be intelligent.



This was a *very* radical idea at that time. The debate that raged then is largely echoed today: Will computers be tools, or will they be conscious entities, having identity, volition, and free will? Turing was a pragmatist. He was just looking for intelligence, not trying to define or evaluate free will. He proposed a test, now called the *Turing Test*, that for him was an intelligence litmus test.

3.1 The Turing Test

The Turing Test is based on the *Imitation Game*, played by three people. In the imitation game, a man and a woman are in one room, and a judge is in the other. The three cannot see one another, so they communicate via Email. The judge questions them for five minutes, trying to discover which of the two is the man and which is the woman. This would be very easy, except that the man lies and pretends to be a woman. The woman tries to help the judge find the truth. If the man is a really good impersonator, he might fool the judge 50% of the time. In practice, it seems the judge is right about 70% of the time.

Now, the Turning Test replaces the man with a computer pretending to be a woman. If the computer can fool the judge 30% of the time, it passes the Turing Test.

2. **The Turing Test:** Build a computer system that wins the imitation game at least 30% of the time.

Turing's actual text on this matter is worth re-reading. What he said was:

"I believe that in about fifty years' time it will be possible, to programme computers, with a storage capacity of about 10^9 , to make them play the imitation game so well that an average interrogator will not have more than 70 per cent chance of making the right identification after five minutes of questioning. The original question, "Can machines think?" I believe to be too meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."

With the benefit of hindsight, Turing's predictions read very well. His technology forecast was astonishingly accurate, if a little pessimistic. The typical computer has the requisite capacity, and is comparably powerful. Turing estimated that the human memory is between 10^{12} and 10^{15} bytes, and the high end of that estimate stands today.

On the other hand, his forecast for machine intelligence was optimistic. Few people characterize computers as intelligent. You can interview ChatterBots on the Internet (<http://www.loebner.net/Prizef/loebner-prize.html>) and judge for yourself. I think they are still a long way from passing the Turing Test. But, there has been enormous progress in the last 50 years, and I expect that eventually a machine will indeed pass the Turing Test. To be more

specific, I think it will happen within the next 50 years because I am persuaded by the argument that we are nearing parity with the storage and computational power of simple brains.

To date, machine-intelligence has been more of a partnership with scientists: a symbiotic relationship. To give some stunning examples of progress in machine intelligence, computers helped with the proofs of several theorems (the four-color problem is the most famous example [9]), and have solved a few open problems in mathematics. It was front page news when IBM's Deep Blue beat the world chess champion. Computers help design almost everything now – they are used in conceptualization, simulation, manufacturing, testing, and evaluation.

In all these roles, computers are acting as tools and collaborators rather than intelligent machines. Vernor Vinge calls this IA (intelligence amplification) as opposed to AI [11]. These computers are not forming new concepts. They are typically executing static programs with very little adaptation or learning. In the best cases, there is a pre-established structure in which parameters automatically converge to optimal settings for this environment. This is adaptation, but it is not learning new things they way a child, or even a spider seems to.

Despite this progress, there is general pessimism about machine intelligence, and artificial intelligence (AI). We are still in AI winter. The AI community promised breakthroughs, but they did not deliver. Enough people have gotten into enough trouble on the Turing Test, that it has given rise to the expression *Turing Tar Pit* “Where everything is possible but nothing is easy.” *AI complete* is short for even harder than NP complete. This is in part a pun on Turing’s most famous contribution: the proof that very simple computers can compute anything computable.

Paradoxically, today it is much easier to research machine intelligence because the machines are so much faster and so much less expensive. This is the “counting argument” that Turing used. Desktop machines should be about as intelligent as a spider or a frog, and supercomputers ought to be nearing human intelligence.

The argument goes as follows. Various experiments and measures indicate that the human brain stores at most 10^{14} bytes (100 Terabytes). The neurons and synaptic fabric can execute about 100 tera-operations per second. This is about thirty times more powerful than the biggest computers today. So, we should start seeing intelligence in these supercomputers any day now (just kidding). Personal computers are a million times slower and 10,000 times smaller than that.

This is similar to the argument that the human genome is about a billion base pairs. 90% of it is junk, 90% of the residue is in common with chimpanzees, and 90% of that residue is in common with all people. So each individual has just a million unique base pairs (and would fit on a floppy disk).

Both these arguments appear to be true. But both indicate that we are missing something *very* fundamental. There is more going on here than we see. Clearly, there is more than a megabyte difference among babies. Clearly, the software and databases we have for our super-computers is not on a track to pass the Turing Test in the next decade. Something quite different is needed. Out-of-the-box, radical thinking is needed.

We have been handed a puzzle: genomes and brains work. But we are clueless what the solution is. Understanding the answer is a wonderful long-term research goal.

3.2. Three more Turing Tests: prosthetic hearing, speech, and vision.

Implicit in the Turing Test, are two sub-challenges that in themselves are quite daunting: (1) read and understand as well as a human, and (2) think and write as well as a human. Both of these appear to be as difficult as the Turing Test itself.

Interestingly, there are three other problems that appear to be easier, but still very difficult: There has been great progress on computers hearing and identifying natural language, music, and other sounds. Speech-to-text systems are now quite useable. Certainly they have benefited from faster and cheaper computers, but the algorithms have also benefited from deeper language understanding, using dictionaries, good natural language parsers, and semantic nets. Progress in this area is steady, and the error rate is dropping about 10% per year. Right now unlimited-vocabulary, continuous speech with a trained speaker and good microphone recognizes about 95% of the words. I joke that computers understand English much better than most people (note: most people do not understand English at all.) Joking aside, many blind, hearing impaired, and disabled people use speech-to-text and text-to-speech systems for reading, listening, or typing.

Speaking as well as a person, given a prepared text, has received less attention than the speech recognition problem, but it is an important way for machines to communicate with people.

There was a major thrust in language translation in the 1950s, but the topic has fallen out of favor. Certainly simple language translation systems exist today. A system that passes the Turing Test in English, will likely have a very rich internal representation. If one teaches such a system a second language, say Mandarin, then the computer would likely have a similar internal representation for information in that language. This opens up the possibility for faithful translation between languages. There may be a more direct path to good language translation, but so far it is not obvious. Bablefish (<http://babelfish.altavista.com/>) is a fair example of the current state of the art. It translates context-free sentences between English and French, German, Italian, Portuguese, and Spanish. It translates the sentence “Please pass the Turing Test” to “Veuillez passer l'essai de Turing”, which translates back to “Please pass the test of Turing.”

The third area, is visual recognition: build a system that can identify objects and recognize dynamic object behaviors in a scene (horse-running, man-smiling, body gestures,...).

Visual rendering is an area where computers already outshine all but the best of us. Again, this is a man-machine symbiosis, but the “special effects” and characters of from Lucasfilm and Pixar are stunning. Still, the challenge remains to make it easy for kids and adults to create such illusions in real time for fun or to communicate ideas.

The Turing Test also suggests prosthetic memory, but I'll reserve that for Bush's section. So the three additional Turing Tests are:

- 3. **Speech to text:** Hear as well as a native speaker.
- 4. **Text to speech:** Speak as well as a native speaker.
- 5. **See as well as a person:** recognize objects and behavior.

As limited as our current progress is in these three areas, it is still a boon to the handicapped and in certain industrial settings. Optical character recognition is used to scan text and speech

synthesizers read the text aloud. Speech recognition systems are used by deaf people to listen to telephone calls and are used by people with carpal tunnel syndrome and other disabilities to enter text and commands. Indeed, some programmers use voice input to do their programming.

For a majority of the deaf, devices that couple directly to the auditory nerve could convert sounds to nerve impulses thereby replacing the eardrum and the cochlea. Unfortunately, nobody yet understands the coding used by the body. But, it seems likely that this problem will be solved someday.

Longer term these prosthetics will help a much wider audience. They will revolutionize the interface between computers and people. When computers can see and hear, it should be much easier and less intrusive to communicate with them. They will also help us to see better, hear better, and remember better.

I hope you agree that these four tests meet the criterion I set out for a good goal, they are understandable, challenging, useful, testable, and they each have incremental steps.

4. Bush's Memex

Vannevar Bush was an early information technologist: he had built analog computers at MIT. During World War II, he ran the Office of Scientific Research and Development. As the war ended he wrote a wonderful piece for the government called the *Endless Frontier* [13][14] that has defined America's science policy for the last fifty years.



In 1945 Bush published a visionary piece "As We May Think" in *The Atlantic Monthly*, <http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm> [14]. In that article, he described *Memex*, a desk that stored "a billion books", newspapers, pamphlets, journals, and other literature, all hyper-linked together. In addition, Bush proposed a set of glasses with an integral camera that would photograph things on demand, and a Dictaphone that would record what was said. All this information was also fed into *Memex*.

Memex could be searched by looking up documents, or by following references from one document to another. In addition, anyone could annotate a document with links, and those annotated documents could be shared among users. Bush realized that finding information in *Memex* would be a challenge, so he postulated "association search", finding documents that matched some similarity criteria.

Bush proposed that the machine should recognize spoken commands, and that it type when spoken to. And, if that was not enough, he casually mentioned that "a direct electrical path to the human nervous system" might be a more efficient and effective way to ask questions and get answers.

Well, 50 years later, *Memex* is almost here. Most scientific literature is online. The scientific literature doubles every 8 years, and most of the last 15 years are online. Much of Turing's work and Bush's articles are online. Most literature is also online, but it is protected by copyright and so not visible to the web.

The [Library of Congress](#) is online and gets more web visitors each day than regular visitors: even though just a tiny part of the library is online. Similarly, the [ACM97](#) conference was recorded and converted to a web site. After one month, five times more people had visited the web site than the original conference. After 18 months, 100,000 people had spent a total of 50,000 hours watching the presentations on the web site. This is substantially more people and time than attendees at the actual event. The site now averages 200 visitors and 100 hours per week.

This is all wonderful, but anyone who has used the web is aware of its limitations: (1) it is hard to find things on the web, and (2) many things you want are not yet on the web. Still, the web is very impressive and comes close to Bush's vision. It is the first place I look for information. Information is increasingly migrating online to cyberspace. Most new information is created online. Today, it is about 50 times less expensive to store 100 letters (1 MB) on magnetic disk, than to store them in a file cabinet (ten cents versus 5 dollars.) Similarly, storing a photo online is about five times less expensive than printing it and storing the photo in a shoebox. Every year, the cost of cyberspace drops while the cost of real space rises.

The second reason for migrating information in cyberspace is that it can be searched by robots. Programs can scan large document collections, and find those that match some predicate. This is faster, cheaper, easier, and more reliable than people searching the documents. These searches

can also be done from anywhere -- a document in England can be easily accessed by someone in Australia.

So, why isn't everything in Cyberspace? Well, the simple answer is that most information is valuable property and currently, cyberspace does not have much respect for property rights. Indeed, the cyberspace culture is that all information should be freely available to anyone anytime. Perhaps the information may come cluttered with advertising, but otherwise it should be free. As a consequence, most information on the web is indeed advertising in one form or another.

There are substantial technical issues in protecting intellectual property, but the really thorny issues revolve around the law (e.g., What protection does each party have under the law given that cyberspace is trans-national?), and around business issues (e.g., What are the economic implications of this change?). The latter two issues are retarding the move of "high value" content to the Internet, and preventing libraries from offering Internet access to their collections. Often, customers must come to the physical library to browse the electronic assets.

Several technical solutions to copy-protect intellectual property are on the table. They all allow the property owner to be paid for use of his property on a per view, or subscription, or time basis. They also allow the viewers and listeners to use the property easily and anonymously. But, until the legal and business issues are resolved, these technical solutions will be of little use.

Perhaps better schemes will be devised that protect intellectual property, but in the mean time we as scientists must work to get our scientific literature online and freely available. Much of it was paid for by taxpayers or corporations, so it should not be locked behind publisher's copyrights. To their credit, our technical society the ACM has taken a very progressive view on web publishing. Your ACM technical articles can be posted on your web site, your department's web site, and on the Computer Science Online Research Repository ([CoRR](#)). I hope other societies will follow ACM's lead on this.

4.1 Personal Memex

Returning to the research challenges, the sixth problem is to build a personal Memex. A box that records everything you see, hear, or read. Of course it must come with some safeguards so that only *you* can get information out of it. But, it should on command, find the relevant event and display it to you. The key thing about this Memex is that it does not do any data analysis or summarization, it just returns what it sees and hears.

6. **Personal Memex:** Record everything a person sees and hears, and quickly retrieve any item on request.

Since it only records what you see and hear, personal Memex seems not to violate any copyright issues [15]. It still raises some difficult ethical issues. If you and I have a private conversation, does your Memex have the right to disclose our conversation to others? Can you sell the conversation without my permission? But, if one takes a very conservative approach: only record with permission and make everything private, then Memex seems within legal bounds. But the designers must be vigilant on these privacy issues.

Memex seems feasible today for everything but video. A personal record of everything you ever read is about 25 GB. Recording everything you hear is a few terabytes. A personal Memex will grow at 250 megabytes (MB) per year to hold the things you read, and 100 gigabytes (GB) per year to hold the things you hear. This is just the capacity of one modern magnetic tape or 2 modern disks. In three years it should be one disk or tape per year. So, if you start recording now, you should be able to stick with one or two tapes for the rest of your life.

Video Memex seems beyond our technology today, but in a few decades, it will likely be economic. High visual quality would be hundreds times more -- 80 terabytes (TB) per year. That is a lot of storage, eight petabytes (PB) per lifetime. It will continue to be more than most individuals can afford. Of course, people may want very high definition and stereo images of what they see. So, this 8 petabyte could easily rise to ten times that. On the other hand, techniques that recognize objects might give huge image compression. To keep the rate to a terabyte a year, the best we can offer with current compression technology is about ten TV-quality frames per second. Each decade the quality will get at least 100x better. Capturing, storing, organizing, and presenting this information is a fascinating long-term research goal.

4.2 World Memex

What about Bush's vision of putting *all* professionally produced information into Memex? Interestingly enough, a book is less than a megabyte of text and all the books and other printed literature is about a petabyte in Unicode. There are about 500,000 movies (most very short). If you record them with DVD quality they come to about a petabyte. If you scanned all the books and other literature in the Library of Congress the images would be a few petabytes. There are 3.5 million sound recordings (most short) which add a few more petabytes. So the consumer-quality digitized contents of the Library of Congress total a few petabytes. Librarians who want to preserve the images and sound want 100x more fidelity in recording and scanning the images, thus getting an exabyte. Recording all TV and radio broadcasts (everywhere) would add 100 PB per year.

Michael Lesk did a nice analysis of the question "How much information is there?" He concludes that there are 10 or 20 exabytes of recorded information (excluding personal and surveillance videotapes) [16]. An interesting fact is that the storage industry shipped exabyte of disk storage in 1999 and about 100 exabytes of tape storage. Near-line (tape) and on-line (disk) storage cost between a 10 k\$ and 100 k\$ per terabyte. Prices are falling faster than Moore's law – storage will likely be a hundred times cheaper in ten years. So, we are getting close to the time when we can record most of what exists very inexpensively. For example, a lifetime cyberspace cemetery plot for your most recent 1 MB research report or photo of your family should cost about 25 cents. That is 10 cents for this year, 5 cents for next year, 5 cents for the successive years, and 5 cents for insurance.

Where does this lead us? If everything will be in cyberspace, how do we find anything? Anyone who has used the web search engines knows both joy and frustration: sometimes they are wonderful and find just what you want. They do some summarization, giving title and first few sentences. But they do very little real analysis or summarization.

So, the next challenge after a personal Memex that just returns exactly what you have seen, undigested, is a Memex that analyzes a large corpus of material and then presents it to you an a convenient way. Raj Reddy described a system that can read a textbook and then answer the

questions at the end of the text as well as a (good) college student [17]. A more demanding task is to take a corpus of text, like the Internet or the Computer Science journals, or Encyclopedia Britannica, and be able to answer summarization questions about it as well as a human expert in that field.

Once we master text, the next obvious step is to build a similar system that can digest a library of sounds (speeches, conversations, music, ...). A third challenge is a system that can absorb and summarize a collection of pictures, movies, and other imagery. The Library of congress has 115 million text and graphic items, the Smithsonian has 140 million items which are 3D (e.g. the Wright Brothers airplane). Moving those items to cyberspace is an interesting challenge. The visible humans (http://www.nlm.nih.gov/research/visible/visible_human.html), millimeter slices versions of two cadavers, give a sense of where this might go. Another exciting project is copying some of Leonardo DeVinci's work to cyberspace.

7. **World Memex:** Build a system that given a text corpus, can answer questions about the text and summarize the text as precisely and quickly as a human expert in that field. Do the same for music, images, art, and cinema.

The challenge in each case is to automatically parse and organize the information. Then when a someone has a question, the question can be posed in a natural interface that combines a language, gesture, graphics, and forms interface. The system should respond with answers which are appropriate to the level of the user.

This is a demanding task. It is probably AI Complete, but it an excellent goal, probably simpler and more useful than a computer that plays the imitation game as well as a human.

4.3 Telepresence

One interesting aspect of being able to record everything is that other people can observe the event, either immediately, or retrospectively. I now routinely listen to lectures recorded at another research institution. Sometimes, these are “live”, but usually they are on-demand. This is extraordinarily convenient -- indeed, many of us find this time-shifting to be even more valuable than space-shifting. But, it is fundamentally just television-on-demand; or if it is audio only, just radio-on-demand – turning the Internet into the world’s most expensive VCR.

A much higher-quality experience is possible with the use of computers and virtual reality. By recording an event in high-fidelity from many angles, computers can reconstruct any the scene at high-fidelity from any perspective. This allows a viewer to sit anywhere in the space, or wander around the space. For a sporting event, the spectator can be on the field watching the action close up. For a business meeting, the participant can sit in the meeting and look about to read facial gestures and body language as the meeting progresses.

The challenge is to record events and then create a virtual environment on demand that allows the observer to experience the event as well as actually being there. This is called **Tele-Observer** because it is really geared to a passive observer of an event – either because it is a past event, or because there are so many observers that they must be passive (they are watching, not interacting). Television and radio give a low-quality version of this today, but they are completely passive.

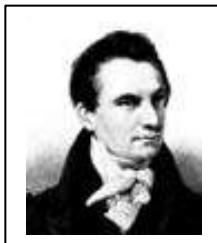
The next challenge is to allow the participant to interact with the other members of the event, i.e. be **Tele-Present**. Tele-presence already exists in the form of telephones, teleconferences, and chat rooms. But, again the experience there is very much lower quality than actually being present. Indeed, people often travel long distances just to get the improved experience. The operational test for Telepresence is that a group of students taking a telepresent class score as well as students who were physically present in the classroom with the instructor. And that the instructor has the same rapport with the telepresent students, as he has with the physically present ones.

8. **TelePresence:** Simulate being some other place retrospectively as an observer (TeleObserver): hear and see as well as actually being there, and as well as a participant, and simulate being some other place as a participant (TelePresent): interacting with others and with the environment as though you are actually there.

There is great interest in allowing a telepresent person to physically interact with the world via a robot. The robot is electrical and mechanical engineering, the rest of the system is information technology. That is why I have left out the robot. As Dave Huffman said: “Computer Science has the patent on the byte and the algorithm. EE has the electron and Physics has energy and matter.”

5. Charles Babbage's Computers

Turing's and Bush's visions are heady stuff: machine intelligence, recording everything, and telepresence. Now it is time to consider long-term research issues for traditional computers. Charles Babbage (1791-1871) had two computer designs, a difference engine, that did numeric computations well, and a fully programmable analytical engine that had punched card programs, a 3-address instruction set, and a memory to hold variables. Babbage loved to compute things and was always looking for trends and patterns. machines to help him do his computations.



He wanted these

By 1955, Babbage's vision of a computer had been realized. Computers with the power he envisioned were generating tables of numbers, were doing bookkeeping, and generally doing what computers do. Certainly there is more to do on Babbage's vision. We need better computational algorithms, and better and faster machines.

But I would like to focus on another aspect of Babbage's computers. What happens when computers become free, infinitely fast, with infinite storage, and infinite bandwidth? Now, this is not likely to happen anytime soon. But, computation has gotten a 10 million times cheaper since 1950 and a trillion times cheaper since 1899 (see Figure1). Indeed, in the last decade they have gotten a thousand times cheaper. So, from the perspective of 1950, computers today are almost free, and have almost infinite speed, storage, and bandwidth.

Figure 1 charts how things have changed since Babbage's time. It measures the price-performance of these systems. Larry Roberts proposed a performance/price metric of

$$\text{Performance / Price} = \frac{\text{Operations Per Second} \cdot \text{Bits Per Operation}}{\text{System Price}}$$

This measures *bits-processed per dollar*. In 1969 Roberts observed that this metric was doubling about every 18 months. This was contemporaneous with Gordon Moore's observation about gates-per silicon chip doubling, but was measured at the system level. Using data from Hans' Moravac's web site (<http://www.frc.ri.cmu.edu/~hpm/book98/>), and some corrections from Gordon Bell, we plotted the data from Herman Hollerith forward. Between 1890 and 1945, systems were either mechanical or electro-mechanical. Their performance doubled about every seven years. In the 1950's, computing shifted to tubes and transistors and the doubling time dropped to 2.3 years. In 1985, microprocessors and VLSI came on the scene. Through a combination of lower systems prices and much faster machines, the doubling time has dropped to one year.

This acceleration in performance/price is astonishing, and it changes the rules. Similar graphs apply to the cost of storage and bandwidth.

This is real deflation. When processing, storage, and transmission cost micro-dollars, then the only real value is the data and its organization. But we computer scientists have some dirty laundry: our "best" programs typically have a bug for every thousand lines of code, and our "free" computers cost at least a thousand dollars a year in care-and-feeding known as system administration.

Computer owners pay comparatively little for them today. A few hundred dollars for a palmtop or desktop computer, a few thousand dollars for a workstation, and perhaps a few tens of thousands for a server. These folks do not want to pay a large operations staff to manage their

systems. Rather, they want a self-organizing system that manages itself. For simple systems like handheld computers, the customer just wants the system to work. Always be up, always store the data, and never lose data. When the system needs repairing, it should “call home” and schedule a fix. Either the replacement system arrives in the mail, or a replacement module arrives in the mail – and no information has been lost. If it is a software or data problem, the software or data is just refreshed from the server in the sky. If you buy a new appliance, you just plug it in and it refreshes from the server in the sky (just as though the old appliance had failed).

This is the vision that most server companies are working towards in building information appliances. You can see prototypes of it by looking at WebTVs or your web browser for example.

5.1. Trouble-Free Systems

So, who manages the server-in-the sky? Server systems are more complex. They have some semi-custom applications, they have much heavier load, and often they provide the very services the hand-held, appliances, and desktops depend on. To some extent, the complexity has not disappeared, it has just moved.

People who own servers do not mind managing the server content, that is their business. But, they do not want to be systems management experts. So, server systems should be self managing. The human systems manager should set goals, polices, and a budget. The system should do the rest. It should distribute work among the servers. When new modules arrive, they should just add to the cluster when they are plugged in. When a server fails, its storage should have been replicated somewhere else, so the storage and computation can move to those new locations. When some hardware breaks, the system should diagnose itself and order replacement modules which arrive by express mail. Hardware and software upgrades should be automatic.

This suggests the very first of the Babbage goals: trouble-free systems.

9. **Trouble-Free Systems:** Build a system used by millions of people each day and yet administered and managed by a single part-time person.

The operational test for this is that it serves millions of people each day, and yet it is managed by a fraction of a person who does all the administrative tasks. Currently, such a system would need 24-hour a day coverage by a substantial staff. With special expertise required for upgrades, maintenance, system growth, database administration, backups, network management, and the like.

5.2. Dependable Systems

Two issues hiding in the previous requirements deserve special attention. There have been a rash of security problems recently: Melissa, Chernobyl, and now a mathematical attack on RSA that makes 512-bit keys seem dangerously small.

We cannot trust our assets to cyberspace if this trend continues. A major challenge for systems designers is to develop a system which only services authorized uses. Service cannot be denied.

Attackers cannot destroy data, nor can they force the system to deny service to authorized users. Moreover, users cannot see data unless they are so authorized.

The added hook here is that most systems are penetrated by stealing passwords and entering as an authorized user. Any authentication based on passwords or other tokens seems too insecure. I believe we will have to go to physio-metric means like retinal scans or some other unforgeable authenticator – and that all software must be signed in an unforgeable way.

The operational test for this research goal is that a tiger team cannot penetrate the system. Unfortunately, that test does not really prove security. So this is one of those instances where the security system must rest on a *proof* that it is secure, and that all the threats are known and are guarded against.

The second attribute is that the system should always be available. We have gone from 90% availability in the 1950s to 99.99% availability today for well managed systems. Web uses experience about 99% availability due to the fragile nature of the web, its protocols, and the current emphasis on time-to-market.

Nonetheless, we have added three 9s in 45 years, or about 15 years per order-of-magnitude improvement in availability. We should aim for five more 9s: an expectation of one second outage in a century. This is an extreme goal, but it seems achievable if hardware is very cheap and bandwidth is very high. One can replicate the services in many places, use transactions to manage the data consistency, use design diversity to avoid common mode failures, and quickly repair nodes when they fail. Again, this is not something you will be able to test: so achieving this goal will require careful analysis and proof.

10. **Secure System:** Assure that the system of problem 9 only services authorized users, service cannot be denied by unauthorized users, and information cannot be stolen (and prove it.)

11. **AlwaysUp:** Assure that the system is unavailable for less than one second per hundred years -- 8 9's of availability (and prove it.)

5.3. Automatic Programming.

This brings us to the final problem: Software is expensive to write. It is the only thing in cyberspace that is getting more expensive, and less reliable. Individual pieces of software are not really less reliable, it is just that the typical program has one bug per thousand lines after it has been tested and retested. The typical software product grows fast, and so adds bugs as it grows.

You might ask how programs could be so expensive? It is simple: designing, creating, and documenting a program costs about 20\$ per line. It costs about 150% of that to test the code. Then once the code is shipped, it costs that much again to support and maintain the code over its lifetime.

This is grim news. As computers become cheaper, there will be more and more programs and this burden will get worse and worse.

The solution so far is to write fewer lines of code by moving to high-level non-procedural languages. There have been some big successes. Code reuse from SAP, PeopleSoft, and others are an enormous savings to large companies building semi-custom applications. The companies still write a lot of code, but only a small fraction of what they would have written otherwise.

The user-written code for many database applications and many web applications is tiny. The tools in these areas are very impressive. Often they are based on a scripting language like JavaScript and a set of pre-built objects. Again an example of software reuse. End users are able to create impressive websites and applications using these tools.

If your problem fits one of these pre-built paradigms, then you are in luck. If not, you are back to programming in C++ or Java and producing a 5 to 50 lines of code a day at a cost of 100\$ per line of code.

So, what is the solution? How can we get past this logjam? Automatic programming has been the Holy Grail of programming languages and systems for the last 45 years. Sad to report, there has been relatively little progress -- perhaps a factor of 10, but certainly not a factor of 1,000 improvement in productivity unless your problem fits one of the application-generator paradigms mentioned earlier.

Perhaps the methodical software-engineering approaches will finally yield fruit, but I am pessimistic. I believe that an entirely new approach is needed. Perhaps it is too soon, because this is a Turing Tar Pit. I believe that we have to (1) have a high level specification language that is a thousand times easier and more powerful than the current languages, (2) computers should be able to compile the language, and (3) the language should be powerful enough so that all applications can be described.

We have systems today that do any two of these three things, but none that do all three. In essence this is the imitation game for a programming staff. The customer comes to the programming staff and describes the application. The staff returns with a proposed design. There is discussion, a prototype is built and there is more discussion. Eventually, the desired application is built.

12. Automatic Programmer: Devise a specification language or user interface that:

- (a) makes it easy for people to express designs (1,000x easier),
- (b) computers can compile, and
- (c) can describe all applications (is complete).

The system should reason about application, asking questions about exception cases and incomplete specification. But it should not be onerous to use.

The operational test is replace the programming staff with a computer, and produce a result that is better and requires no more time than dealing with a typical human staff. Yes, it will be a while until we have such a system, but if Alan Turing was right about machine intelligence, it's just be a matter of time.

6. Summary

These are a dozen very interesting research problems. Each is a long-term research problem. Now you can see why I want the government to invest in long-term research. I suspect that in 50 years future generations of computer scientists will have made substantial progress on each of these problems. Paradoxically, many (5) of the dozen problems appear to require machine intelligence as envisioned by Alan Turing.

The problems fall in the three broad categories: Turing's intelligent machines improving the human-computer interface, Bush's Memex recording, analyzing, and summarizing everything that happens, and Babbage's computers which will finally be civilized so that they program themselves, never fail, and are safe.

No matter how it turns out, I am sure it will be very exciting. As I said at the beginning, progress appears to be accelerating: the base-technology progress in the next 18 months will equal all previous progress, if Moore's law holds. And there are lots more doublings after that.

A Dozen Long-Term Systems Research Problems.

1. **Scalability:** Devise a software and hardware architecture that scales up by a factor for 10^6 . That is, an application's storage and processing capacity can automatically grow by a factor of a million, doing jobs faster (10^6 x speedup) or doing 10^6 larger jobs in the same time (10^6 x scaleup), just by adding more resources.
2. **The Turing Test:** Build a computer system that wins the imitation game at least 30% of the time.
3. **Speech to text:** Hear as well as a native speaker.
4. **Text to speech:** Speak as well as a native speaker.
5. **See as well as a person:** recognize objects and motion.
6. **Personal Memex:** Record everything a person sees and hears, and quickly retrieve any item on request.
7. **World Memex:** Build a system that given a text corpus, can answer questions about the text and summarize the text as precisely and quickly as a human expert in that field. Do the same for music, images, art, and cinema.
8. **TelePresence:** Simulate being some other place retrospectively as an observer (TeleOberserver): hear and see as well as actually being there, and as well as a participant, and simulate being some other place as a participant (TelePresent): interacting with others and with the environment as though you are actually there.
9. **Trouble-Free Systems:** Build a system used by millions of people each day and yet administered and managed by a single part-time person.
10. **Secure System:** Assure that the system of problem 9 only services authorized users, service cannot be denied by unauthorized users, and information cannot be stolen (and prove it).
11. **AlwaysUp:** Assure that the system is unavailable for less than one second per hundred years -- 8 9's of availability (and prove it).
12. **Automatic Programmer:** Devise a specification language or user interface that:
 - (a) makes it easy for people to express designs (1,000x easier),
 - (b) computers can compile, and
 - (c) can describe all applications (is complete).The system should reason about application, asking questions about exception cases and incomplete specification. But it should not be onerous to use.

7. References

- [1] Graph based on data in Hans P. Moravec *Robot, Mere Machines to Transcendent Mind*, Oxford, 1999, ISBN 0-19-511630-5, (<http://www.frc.ri.cmu.edu/~hpm/book98/>) personal communication with Larry Roberts who developed the metric in 1969, and personal communication with Gordon Bell who helped analyze the data and corrected some errors.
- [2] CSTB–NRC, *Evolving the High-Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure*, National Academy Press, Washington DC, 1995.
- [3] CSTB–NRC *Funding a Revolution, Government Support for Computing Research*, National Academy Press, Washington DC, 1999. ISBN 0-309-6278-0.
- [4] *Information Technology Research: Investing in Our Future, President's Information Technology Advisory Committee, Report to the President, Feb. 1999*. National Coordination Office for Computing, Information, and Communications, Arlington VA.
- [5] *Donald E. Stokes, Pasteur's Quadrant: Basic Science and Technological Innovation*, Brookings, 1997, ISBN 0-8157-8178-4.
- [6] Stephen E. Ambrose, *Undaunted Courage: Meriwether Lewis, Thomas Jefferson, and the Opening of the American West*, Simon & Schuster, NY, 1996, ISBN: 0684811073
- [7] “From Micro-device to Smart Dust”, *Science News*, 6/26/97, Vol. 152(4), pp 62-63
- [8] Alan Newell, “Fairy Tales,” appears in R. Kruzelweil, *The Age of Intelligent Machines*, MIT Press, 1990, ISBN: 0262610795, pp 420-423
- [9] Alan M. Turing, “Computing Machinery and Intelligence”, *Mind*, Vol. LIX. 433-460, 1950). Also on the web at many sites. K. Appel and W. Haken, “The solution of the four-color-map problem,” *Scientific American*, Oct 1977, 108-121,
<http://www.math.gatech.edu/~thomas/FC/fourcolor.html> (1995) has a “manual” proof
- [11] Vernor Vinge, “Technological Singularity.” VISION-21 Symposium sponsored by NASA Lewis Research Center and the Ohio Aerospace Institute, March, 1993. also at
<http://www.frc.ri.cmu.edu/~hpm/book98/com.ch1/vinge.singularity.html>
- [12] *Endless Frontier: Vannevar Bush, Engineer of the American Century*, G. Pascal Zachary, Free Press, 1997 ISBN: 0-684-82821-9
- [13] Vannevar Bush, *Science-The Endless Frontier*, Appendix 3, "Report of the Committee on Science and the Public Welfare," Washington, D.C.: U.S. Government Printing Office, 1945. Reprinted as National Science Foundation Report 1990, online
http://rits.stanford.edu/siliconhistory/Bush/Bush_text.html
- [14] Vannevar Bush ”As We May Think” *The Atlantic Monthly*, July 1945,
<http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm>
- [15] Anne Wells-Branscomb, *Who Owns Information?: From Privacy to Public Access* Basic Books, 1995, ISBN: 046509144X.
- [16] Micheal Lesk, “How much information is there in the world?”
<http://www.lesk.com/mlesk/ksg97/ksg.html>
- [17] Raj Reddy, “To Dream The Possible Dream,” CACM, May 1996, Vol. 39, No. 5 pp106-112.

Principles for Computer System Design

Butler Lampson

We have learned depressingly little in the last ten years about how to build computer systems. But we have learned something about how to do the job more precisely, by writing more precise specifications, and by showing more precisely that an implementation meets its specification. Methods for doing this are of both intellectual and practical interest. I will explain the most useful such method and illustrate it with two examples:

Connection establishment: Sending a reliable message over an unreliable network.

Transactions: Making a large atomic action out of a sequence of small ones.

Principles for Computer System Design

10 years ago: *Hints for Computer System Design*

Not that much learned since then—disappointing

Instead of standing on each other's shoulders, we stand on each other's toes. (Hamming)

One new thing: How to build systems more precisely

If you think systems are expensive, try chaos.

Collaborators

Bob Taylor

Chuck Thacker

Workstations: Alto, Dorado, Firefly
Networks: AN1, AN2

Charles Simonyi

Bravo WYSIWYG editor

Nancy Lynch

Reliable messages

Howard Sturgis

Transactions

Martin Abadi

Security

Mike Burrows

Morrie Gasser

Andy Goldstein

Charlie Kaufman

Ted Wobber

From Interfaces to Specifications

Make modularity precise

Divide and conquer (Roman motto)

Design

Correctness

Documentation

Do it recursively

Any idea is better when made recursive (Randell)

Refinement: One man's implementation is another man's spec.
(adapted from Perlis)

Composition: Use actions from one spec in another.

Specifying a System with State

A safety property: nothing bad ever happens

Defined by a state machine:

state: a set of values, usually divided into named *variables*

actions: named changes in the state

A liveness property: something good eventually happens

These define behavior: all the possible sequence of actions

Examples of systems with state:

Data abstractions

Concurrent systems

Distributed systems

You can't observe the actual state of the system from outside.
All you can see is the results of actions.

Editable Formatted Text

State

text: sequence of (Char, Property)

H	e	l	l	o
----------	---	---	---	---



Actions

get(2) returns ('e', (Times-Roman, ...))

replace(3, 5, 2, 3, [a p p l e])

a	p	p	l	e
---	---	---	---	---

H	e	l	p	
----------	---	---	----------	--

H	e	l	l	o
---	---	---	----------	---

look(0, 5, italic := true)

H	e	l	l	o
----------	---	---	---	---

This interface was used in the Bravo editor.
The implementation was about 20k lines of code.

How to Write a Spec

Figure out what the state is

Choose it to make the spec clear, not to match the code.

Describe the actions

What they do to the state

What they return

Helpful hints

Notation is important; it helps you to think about what's going on.

Invent a suitable vocabulary.

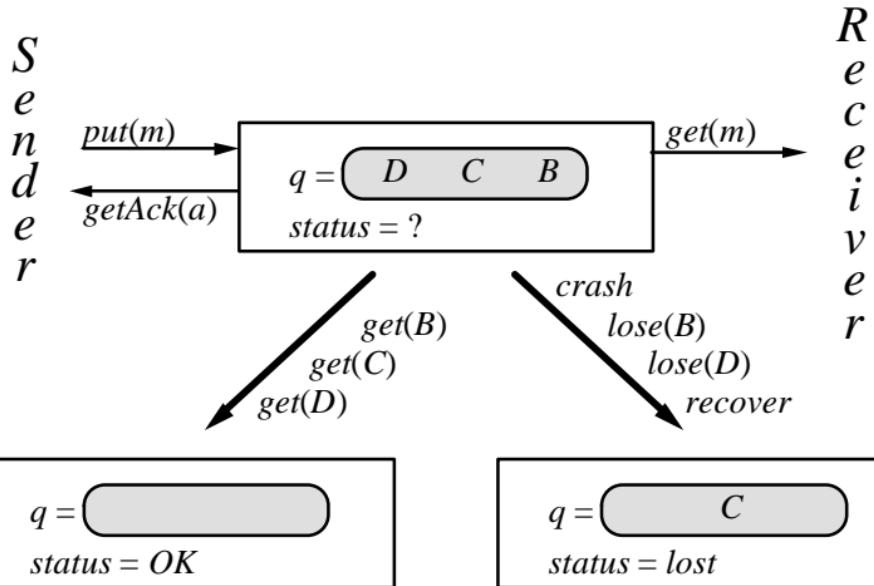
Fewer actions are better.

Less is more.

More non-determinism is better; it allows more implementations.

I'm sorry I wrote you such a long letter; I didn't have time to write a short one.
(Pascal)

Reliable Messages



Spec for Reliable Messages

q : sequence[M] := $<>$
 $status$: {OK, lost, ?} := lost
 $rec_{s/r}$: Boolean := false (short for ‘recovering’)

Name	Guard	Effect	Name	Guard	Effect
$**put(m)$		append m to q , $status := ?$	$*get(m)$	m first on q	remove head of q , if $q = <>$, $status = ?$ then $status := OK$
$*getAck(a)$	$status = a$	$status := lost$	$lose$	rec_s or	delete some element from q ;
	rec_r	if it’s the last then $status := lost$, or $status := lost$			

What “Implements” Means?

Divide actions into *external* and *internal*.

Y implements X if

every external behavior of Y is an external behavior of X, and

Y’s liveness property implies X’s liveness property.

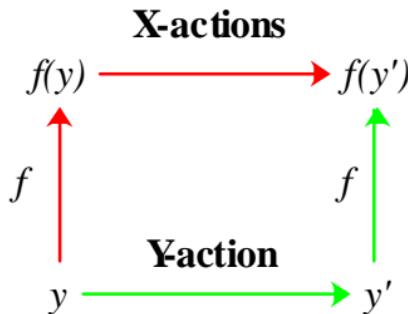
This expresses the idea that Y implements X if
you can’t tell Y apart from X by looking only at the external actions.

Proving that Y implements X

Define an *abstraction function* f from the state of Y to the state of X.

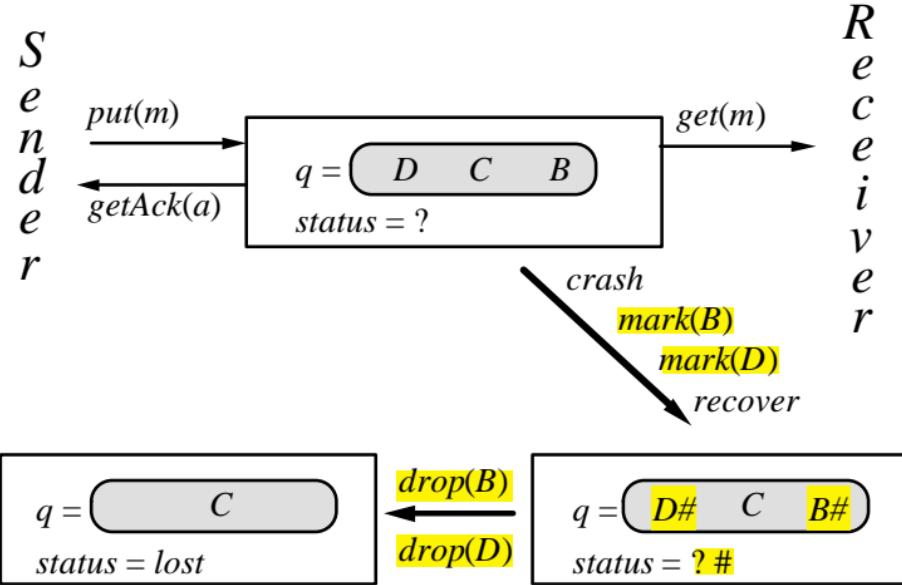
Show that Y *simulates* X:

- 1) f maps initial states of Y to initial states of X.
- 2) For each Y-action and each state y
there is a sequence of X-actions that is the same externally,
such that the diagram commutes.



This always works!

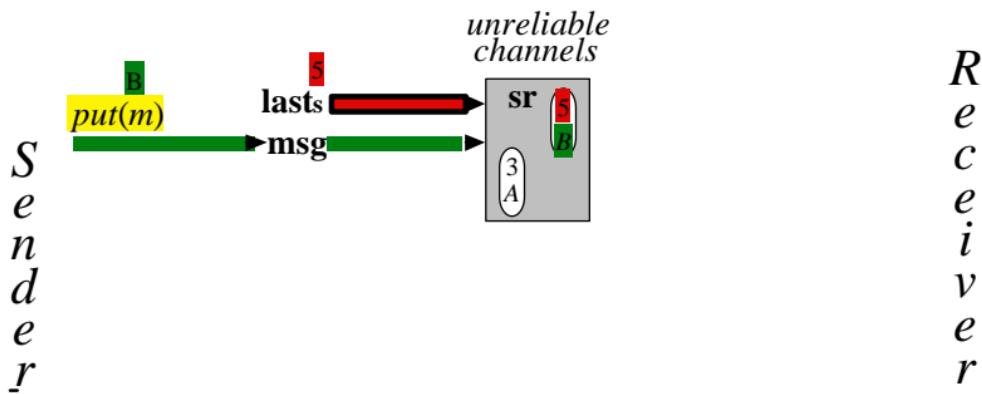
Delayed-Decision Spec: Example



The implementer wants the spec as non-deterministic as possible, to give him more freedom and make it easier to show correctness.

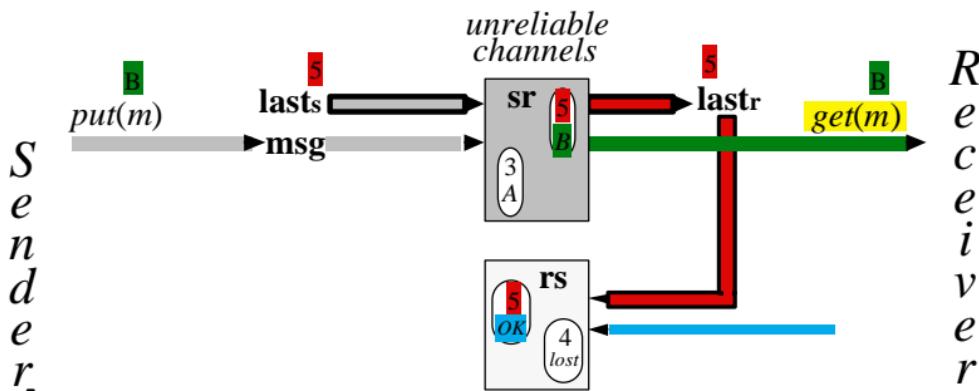
A Generic Protocol G (1)

Sender		Receiver	
actions	state	state	actions

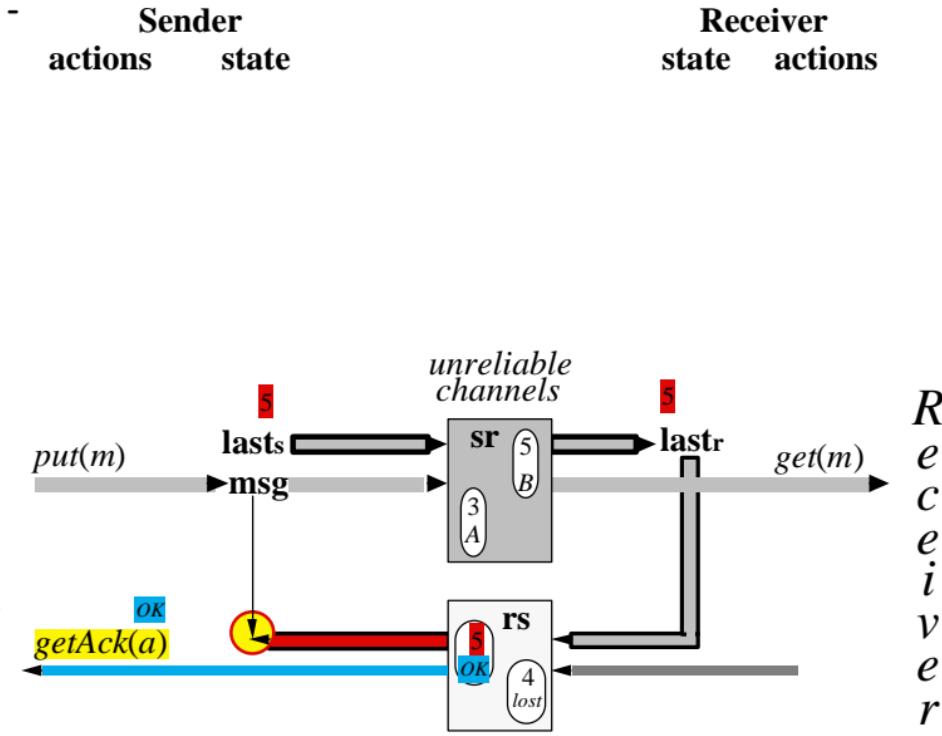


A Generic Protocol G (2)

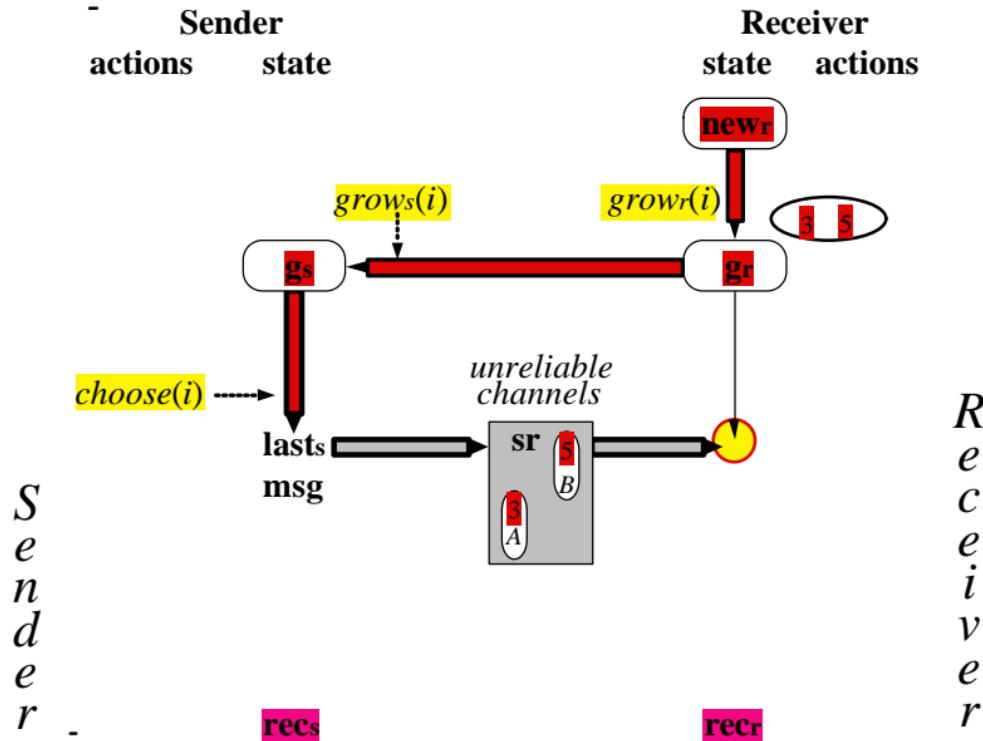
Sender		Receiver	
actions	state	state	actions



A Generic Protocol G (3)

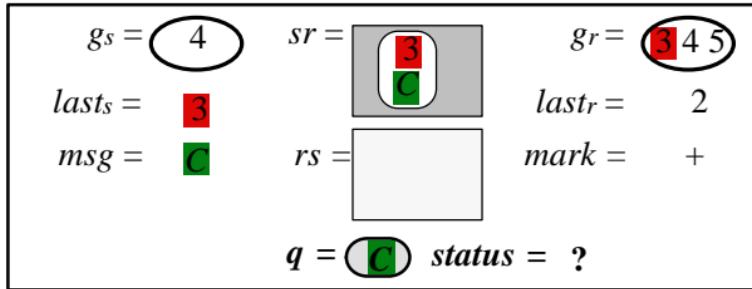


A Generic Protocol G (4)



G at Work

Sender

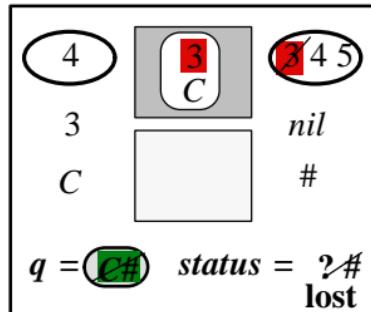
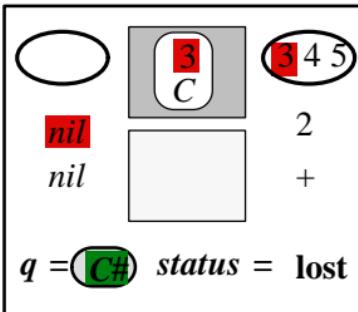
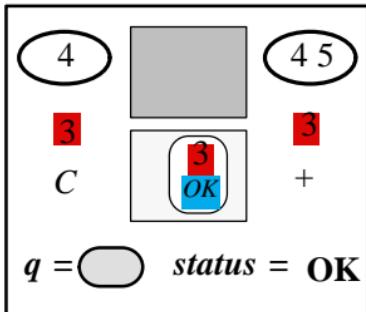


Receiver

$get(C)$

$crashs$

$crashr; recover$
(before strikeout)
 $shrink_r(3)$
(after strikeout)



Abstraction Function for G

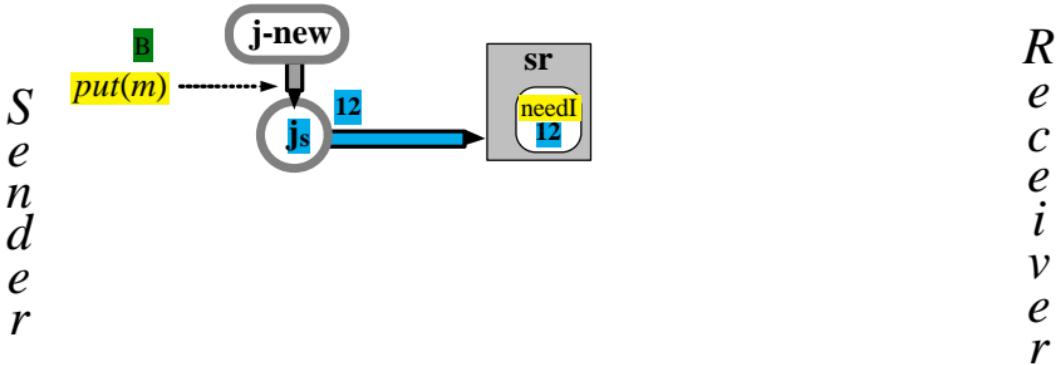
$cur\text{-}q$ $= \begin{cases} <msg> & \text{if } msg = nil \text{ and } (last_s = nil \text{ or } last_s \in gr) \\ <> & \text{otherwise} \end{cases}$
 $old\text{-}q$ $= \text{the messages in } sr \text{ with } i\text{'s that are good and not } = last_s$

q $old\text{-}q + cur\text{-}q$

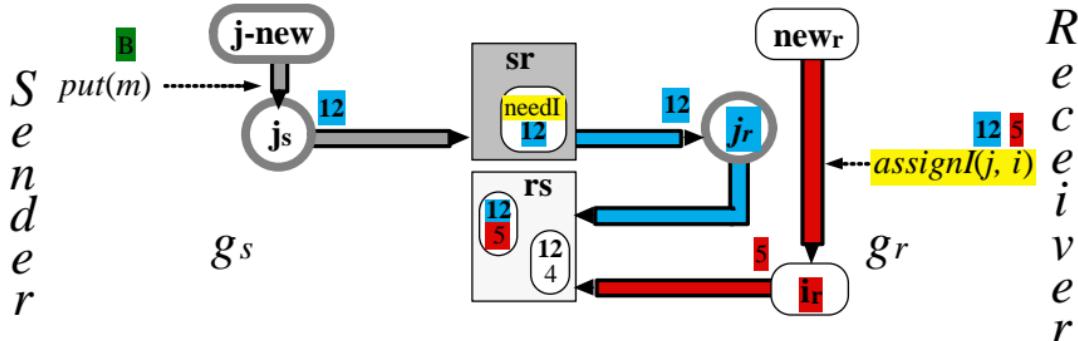
status \square $\begin{cases} ? & \text{if } cur\text{-}q <> \\ OK & \text{if } last_s = last_r = nil \\ lost & \text{if } last_s \notin (gr \cup \{last_r\}) \text{ or } last_s = nil \end{cases}$

rec_{s/r} $rec_{s/r}$

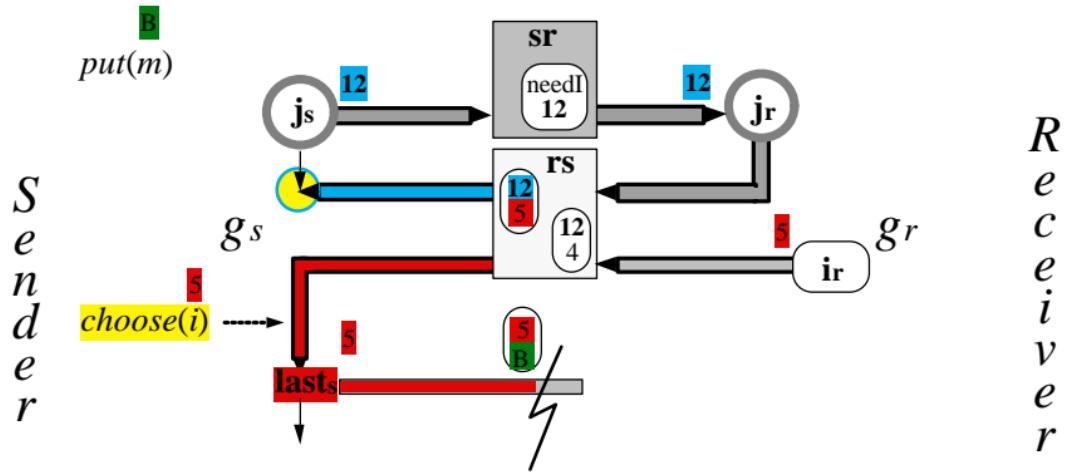
The Handshake Protocol H (1)



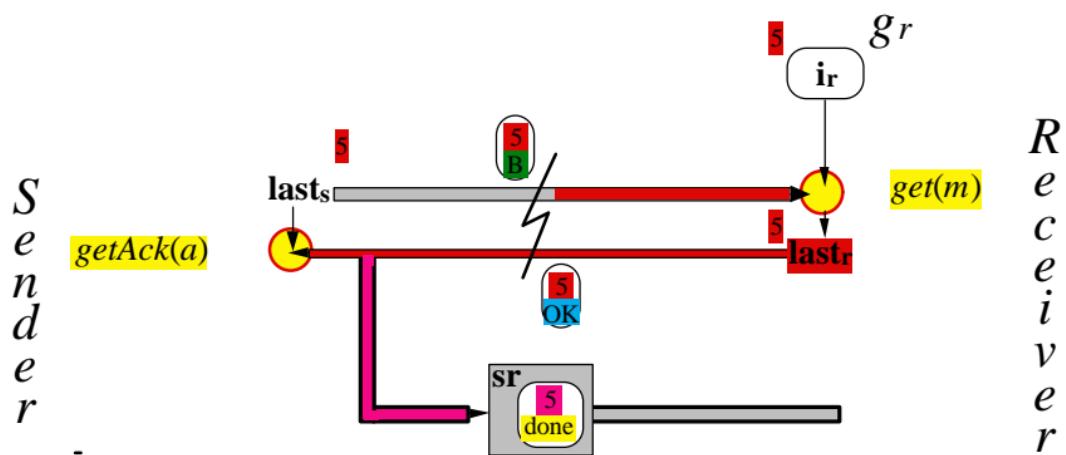
The Handshake Protocol H (2)



The Handshake Protocol H (3)

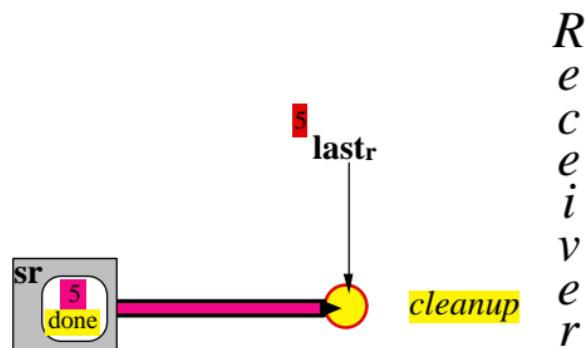


The Handshake Protocol H (4)

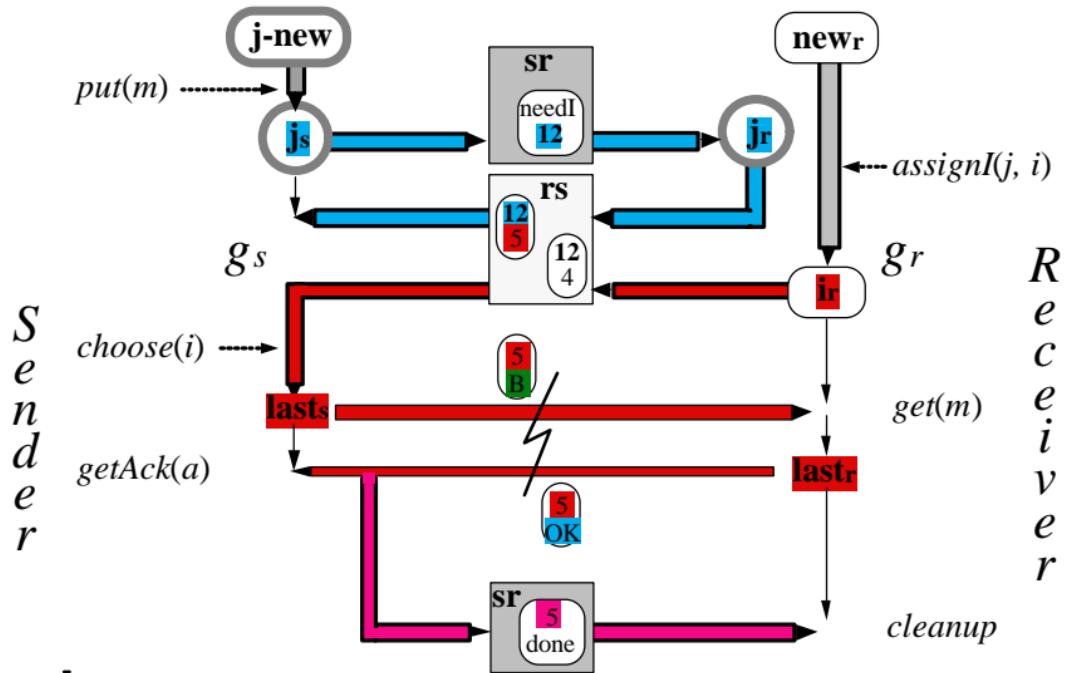


The Handshake Protocol H (5)

S
e
n
d
e
r



The Handshake Protocol H (6)



Abstraction Function for H

G

g_s

H

the i 's with (j_s, i) in rs

g_r

$\{i_r\} - \{\text{nil}\}$

sr and rs the (I, M) and (I, A) messages in sr and rs

$new_{s/r}$, $last_{s/r}$, and msg are the same in G and H

$grow_r(i)$ receiver sets i_r to an identifier from new_r

$grow_s(i)$ receiver sends (j_s, i)

$shrink_s(i)$ channel rs loses the last copy of (j_s, i)

$shrink_r(i)$ receiver gets (i_r, done)

An efficient program is an exercise in logical brinksmanship.
(Dijkstra)

Reliable Messages: Summary

Ideas

Identifiers on messages

Sets of good identifiers, sender's \subseteq receiver's

Cleanup

The spec is simple.

Implementations are subtle because of crashes.

The abstraction functions reveal their secrets.

The subtlety can be factored in a precise way.

Atomic Actions

S : State

Name	Guard	Effect
$do(a):Val$		$(S, val) := a(S)$

X	Y
5	5
$do(x := x - 1)$	
4	5
$do(y := y + 1)$	
4	6

A distributed system is a system in which I can't get my work done because a computer has failed that I've never even heard of.

(Lamport)

Transactions: One Action at a Time

S , s : *State*

Name	Guard	Effect
$do(a):Val$		$(s, val) := a(s)$
$commit$		$S := s$
$crash$		$s := S$

X	Y	x	y
5	5	5	5
$do(x := x - 1); do(y := y + 1)$			
5	5	4	6
<i>commit</i>			
4	6	4	6
<i>crash before commit</i>			
5	5	5	5



Server Failures

$S, s : State$
 $\phi : \{\text{nil}, \text{run}\} := \text{nil}$

Name	Guard	Effect
<i>begin</i>	$\phi = \text{nil}$	$\phi := \text{run}$
<i>do(a):Va</i>	$\phi = \text{run}$	$(s, val) := a(s)$
<i>l</i>		
<i>commit</i>	$\phi = \text{run}$	$S := s, \phi := \text{nil}$
<i>crash</i>		$s := S, \phi := \text{nil}$

Note that we clean up the auxiliary state ϕ .

X	Y	x	y	ϕ
5	5	5	5	nil
<i>do(x := x-1); do(y := y+1)</i>				
5	5	4	6	run
<i>commit</i>				
4	6	4	6	nil
<i>-----</i>				
<i>crash before commit</i>				nil
5	5	5	5	

Incremental State Changes: Logs (1)

$S, s : State$

$L, l : SEQ\ Action := <>$

$\phi : \{\text{nil}, \text{run}\} := \text{nil}$

$S = S + L$

$s, \phi = s, \phi$

Name	Guard	Effect
<i>begin</i>	$\phi = \text{nil}$	$\phi := \text{run}$
<i>do(a):Val</i>	$\phi = \text{run}$	$(s, val) := a(s), l += a$
<i>commit</i>	$\phi = \text{run}$	$L := l, \phi := \text{nil}$
...		
<i>crash</i>		$l := L, s := S+L, \phi := \text{nil}$

X	Y	x	y	Logs	ϕ
5	5	5	5		nil
begin; do(x:=x-1); do(y:=y+1)					
5	5	4	6	$x := 4^*$	run
				$y := 6^*$	
commit					
5	5	4	6	$x := 4^*$	nil
				$y := 6^*$	
<hr/>					
crash before commit					
5	5	5	5		nil

Incremental State Changes: Logs (2)

$S, s : State$

$L, l : SEQ\ Action$

$\phi : \{\text{nil}, \text{run}\}$

$S = S + L$

$s, \phi = s, \phi$

Name	Guard	Effect
		<i>begin, do, and commit as before</i>
<i>apply(a)</i>	$a = \text{head}(l)$	$S := S + a, l := \text{tail}(l)$
<i>cleanLog</i>	L in S	$L := <>$
<i>crash</i>		$l := L, s := S + L, \phi := \text{nil}$

X	Y	x	y	Logs	ϕ
5	5	4	6	$x := 4^*$ $y := 6^*$	nil
				<i>apply(x := 4)</i>	
4	5	"		$x := 4$ $y := 6^*$	nil
				<i>apply(y := 6)</i>	
4	6	"		$x := 4$ $y := 6$	nil
				<i>cleanLog</i>	
4	6	"			nil

<i>crash after apply(x := 4)</i>					
4	5	"		$x := 4^*$ $y := 6^*$	nil

Incremental Log Changes

$S, s : State$

$L, l : SEQ\ Action$

$\Phi, \phi : \{nil, run^*, commit\}$

$L = L \text{ if } \phi = \text{com else } <>$

$\phi = \phi \text{ if } \phi = \text{com else nil}$

Name	Guard	Effect
<i>begin</i> and do as before		
<i>flush</i>	$\phi = \text{run}$	copy some of l to L
<i>commit</i>	$\phi = \text{run}, L = l$	$\Phi := \phi := \text{commit}$
<i>apply(a)</i>	$\phi = \text{commit}, "$	"
<i>cleanLog</i>	head(L) in S	$L := \text{tail}(L)$
	or $\phi = \text{nil}$	
<i>cleanup</i>	$L = <>$	$\Phi := \phi := \text{nil}$
<i>crash</i>		$l := <> \text{ if } \Phi = \text{nil} \text{ else } L;$ $s := S + l, \phi := \Phi$

X	Y	x	y	Logs	Φ	ϕ
5	5	4	6	$x := 4^*$ $y := 6^*$	nil	run
					flush; commit	
5	5	"		$x := 4^*$ $y := 6^*$	com com	
					apply($x := 4$); apply($y := 6$)	
4	6	"		$x := 4$ $y := 6$	com com	
					cleanLog; cleanup	
4	6	"			nil	nil

					crash after flush	
4	5	"		$x := 4^*$ $y := 6^*$	nil	nil

Distributed State and Log

S_i , s_i : State	$\phi = \text{run}$ if all $\phi_i = \text{run}$
L_i , l_i : SEQ Action	com if any $\phi_i = \text{com}$
Φ_i , ϕ_i : {nil, run*, commit}	and any $L_i < >$
S, L, Φ are the products of the S_i, L_i, Φ_i	nil otherwise

Name	Guard	Effect
<i>begin</i> and <i>do</i> as before		
$flush_i$	$\phi_i = \text{run}$	copy some of l_i to L_i
$prepare_i$	$\phi_i = \text{run}$, $L_i = l_i$	$\Phi_i := \text{run}$
$commit$	$\phi = \text{run}$, $L = l$	some $\Phi_i := \phi_i := \text{commit}$
<i>cleanLog</i> and <i>cleanup</i> as before		
$crash_i$		$l_i := < >$ if $\Phi_i = \text{nil}$ else L_i ; $s_i := S_i + l_i$, $\phi_i := \Phi_i$

High Availability

The $\Phi = \text{commit}$ is a possible single point of failure.

With the usual two-phase commit (2PC) this is indeed a limitation on availability.

If data is replicated, an unreplicated commit is a weakness.

Deal with this by using a highly available *consensus algorithm* for Φ .

Lamport's Paxos algorithm is the best currently known.

Transactions: Summary

Ideas

Logs

Commit records

Stable writes at critical points: prepare and commit

Lazy cleanup

The spec is simple.

Implementations are subtle because of crashes.

The abstraction functions reveal their secrets.

The subtlety can be added one step at a time.

How to Write a Spec

Figure out what the state is

Choose it to make the spec clear, not to match the code.

Describe the actions

What they do to the state

What they return

Helpful hints

Notation is important; it helps you to think about what's going on.

Invent a suitable vocabulary.

Fewer actions are better.

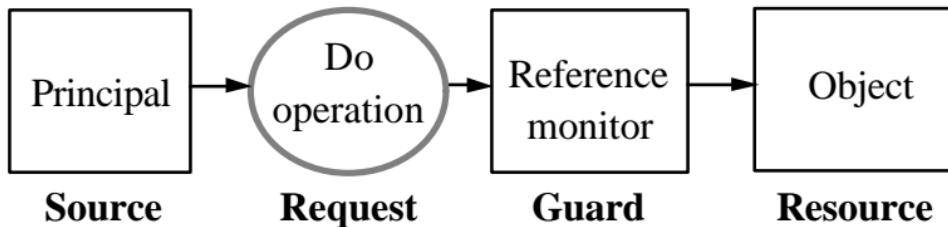
Less is more.

More non-determinism is better; it allows more implementations.

I'm sorry I wrote you such a long letter; I didn't have time to write a short one.
(Pascal)

Security: The Access Control Model

Guards control access to valued resources.

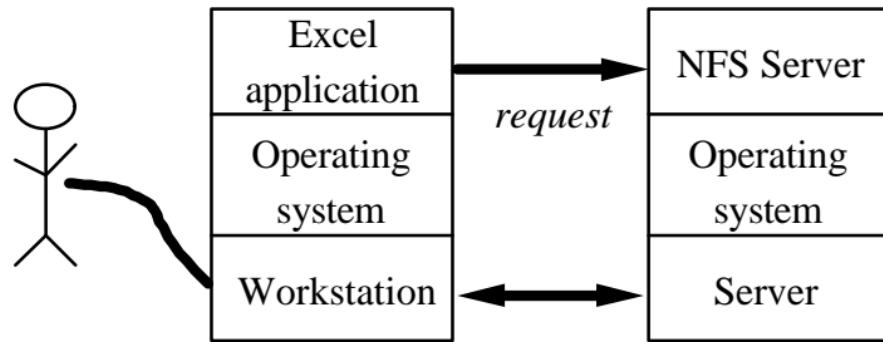


Rules control the operations allowed

for each principal and object.

<i>Principal</i> may do	<i>Operation</i> on	<i>Object</i>
Taylor	Read	File “Raises”
Jones	Pay invoice 4325	Account Q34
Schwarzkopf	Fire three rounds	Bow gun

A Distributed System



Principals

Authentication: Who sent a message?

Authorization: Who is trusted?

Principal — abstraction of "who":

People Lampson, Taylor

Machines VaxSN12648, Jumbo

Services SRC-NFS, X-server

Groups SRC, DEC-Employees

Channels Key #7438

Theory of Principals

Principal says statement

$P \text{ says } s$

Lampson says “read /SRC/Lampson/foo”

SRC-CA says “Lampson’s key is #7438”

Principal A speaks for B

$A \Rightarrow B$

If A says something, B says it too. So A is stronger than B .

A secure channel:

says things directly

$C \text{ says } s$

If P is the only sender on C

$C \Rightarrow P$

Examples

Lampson \Rightarrow SRC

Key #7438 \Rightarrow Lampson

Handing Off Authority

Handoff rule: If A says $B \Rightarrow A$ then $B \Rightarrow A$

Reasonable if A is competent and accessible.

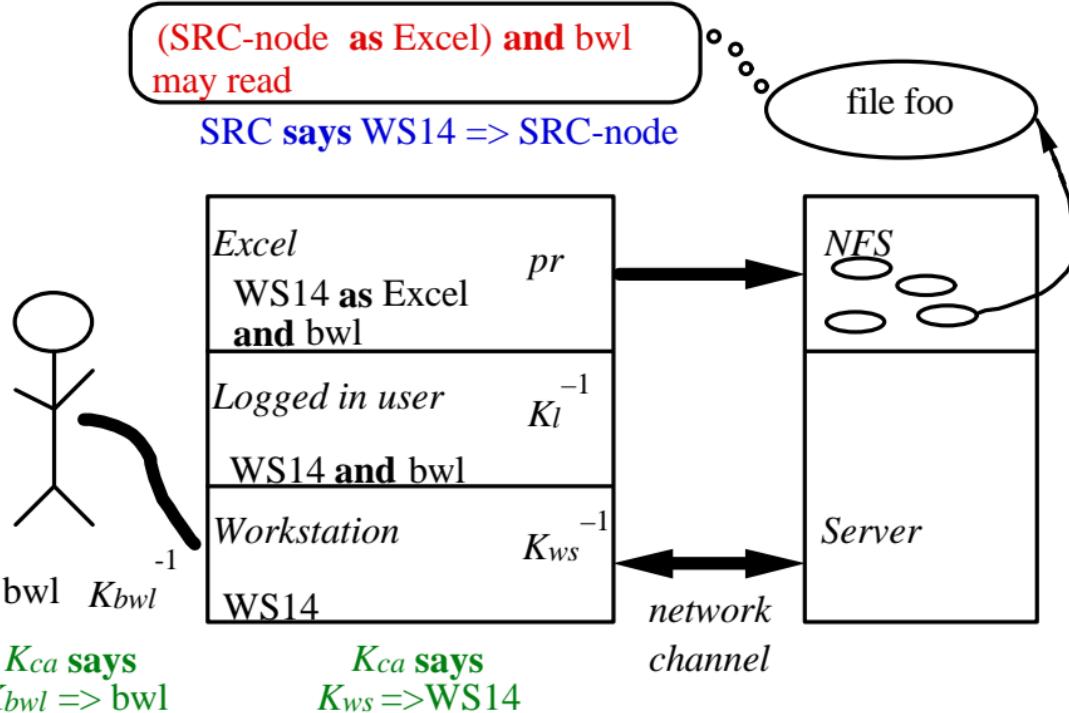
Examples:

SRC says Lampson \Rightarrow SRC

Node key says Channel key \Rightarrow Node key

*Any problem in computer science can be solved
with another level of indirection. (Wheeler).*

Authenticating to the Server



Access Control

Checking access:

Given a request Q says read O
 an ACL P may read O

Check that Q speaks for P
$$Q \Rightarrow P$$

Auditing

Each step is justified by
a signed statement, or
a rule

Authenticating a Channel

Authentication — who can send on a channel.

$C \Rightarrow P$; C is the channel, P the sender.

To get new $C \Rightarrow P$ facts, must trust some principal, a *certification authority*, to tell them to you.

Simplest: trust K_{ca} to authenticate any name:

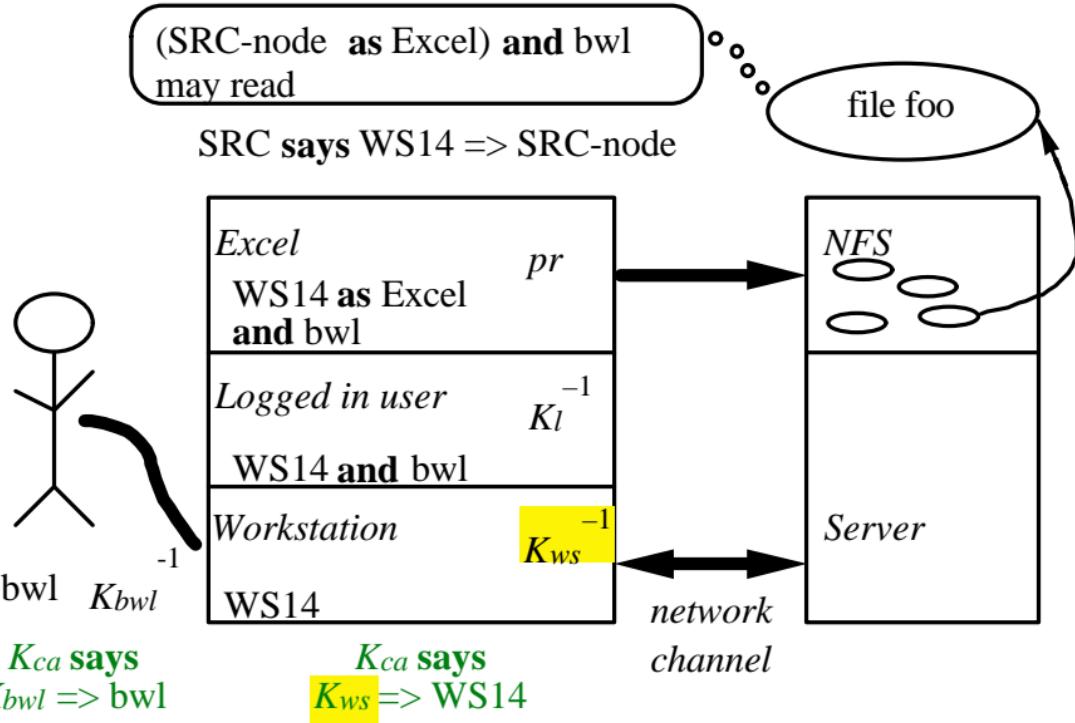
$K_{ca} \Rightarrow \text{Anybody}$

Then CA can authenticate channels:

K_{ca} says K_{ws} \Rightarrow WS

K_{ca} says K_{bwl} \Rightarrow bwl

Authenticated Channels: Example



Groups and Group Credentials

Defining groups: A group is a principal; its members speak for it.

Lampson=> SRC

Taylor => SRC

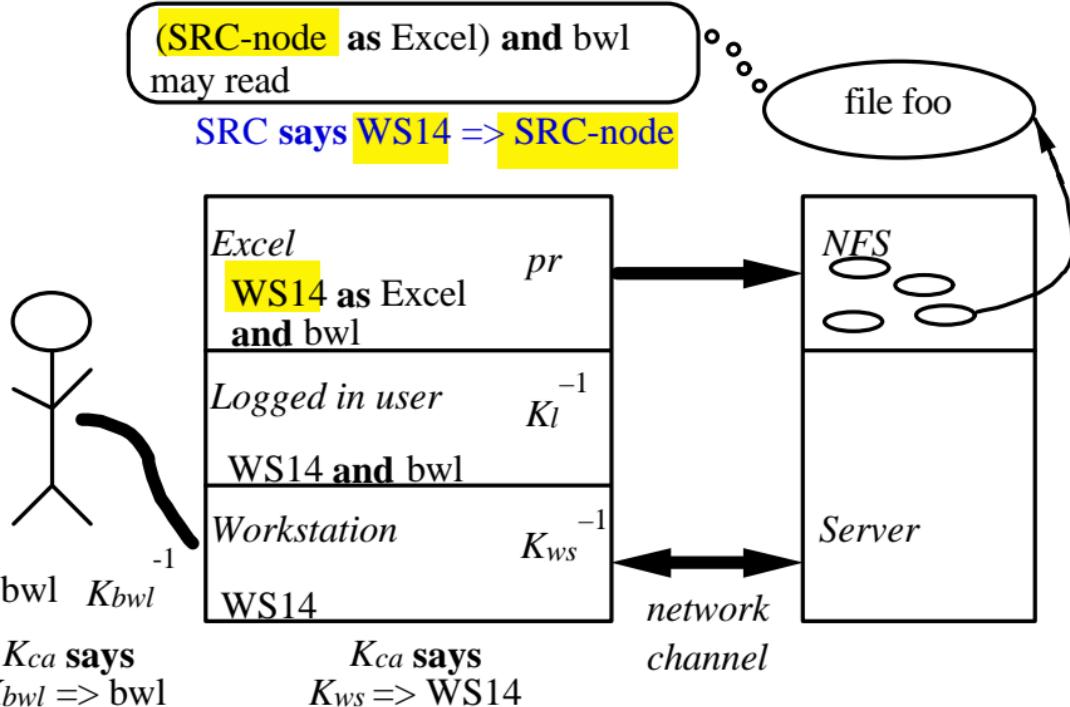
...

Proving group membership: Use certificates.

K_{src} says Lampson => SRC

K_{ca} says K_{src} => SRC

Authenticating a Group



Security: Summary

Ideas

Principals

Channels as principals

“Speaks for” relation

Handoff of authority

Give precise rules.

Apply them to cover many cases.

References

- Hints* Lampson, Hints for Computer System Design.
IEEE Software, Jan. 1984.
- Specifications* Lamport, A simple approach to specifying concurrent systems. *Communications of the ACM*, Jan. 1989.
- Reliable messages* in Mullender, ed., *Distributed Systems*, Addison-Wesley, 1993 (summer)
- Transactions* Gray and Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.
- Security* Lampson, Abadi, Burrows, and Wobber, Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, Nov. 1992.

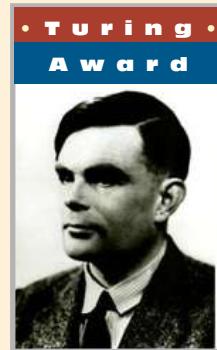
Collaborators

Charles Simonyi	Bravo: WYSIWYG editor
Bob Sproull	Alto operating system Dover: laser printer Interpress: page description language
Mel Pirtle	940 project, Berkeley Computer Corp.
Peter Deutsch	940 operating system QSPL: system programming language
Chuck Geschke	Mesa: system programming language
Jim Mitchell	
Ed Satterthwaite	
Jim Horning	Euclid: verifiable programming language
Ron Rider	Ears: laser printer
Gary Starkweather	
Severo Ornstein	Dover: laser printer

Collaborators

Roy Levin	Wildflower: Star workstation prototype Vesta: software configuration
Andrew Birrell, Roger Needham, Mike Schroeder	Global name service and authentication
Eric Schmidt	System models: software configuration
Rod Burstall	Pebble: polymorphic typed language

Raj Reddy



To Dream The Possible Dream

It is an honor and a pleasure for me to accept this award from ACM. It is especially gratifying to share this award with Ed Feigenbaum, who has been a close friend and helpful colleague for nearly 30 years.

As a second-generation artificial intelligence (AI) researcher, I was fortunate to have known and worked with many of the founding fathers of AI. By observing John McCarthy, my thesis advisor, during the golden age of AI Labs at Stanford in the 1960s, I have learned the importance of fostering and nurturing diversity in research far beyond one's own personal research agenda. Although his own primary interest was in common-sense reasoning and epistemology, under John's leadership, research in speech, vision, robotics, language, knowledge systems, game playing, and music, thrived at the AI labs. In addition, a great deal of path-breaking systems research flourished in areas such as Lisp, time-sharing, video displays, and a precursor to Windows called "pieces of glass."

From Marvin Minsky, who was visiting Stanford and helping to build the Mars Rover in '66, I learned the importance of pursuing bold visions of the future. And from Allen Newell and Herb Simon, my colleagues and mentors at Carnegie Mellon University (CMU) for over 20 years, I learned how one can turn bold visions into practical reality by careful design of experiments and following the scientific method.

I was also fortunate to have known and worked with Alan Perlis, a giant in the '50s and '60s computing scene and the first recipient of the Turing Award in 1966, presented at the ACM conference held in Los Angeles, which I attended while still a graduate student at Stanford.

While I did not know Alan Turing, I may be one of the select few here who used a computer designed by him. In the late 1950s, I had the pleasure of using a mercury delay-line computer (English Electric Deuce Mark II) based on Turing's

original design of ACE. Given his early papers on "Intelligent Machines," Turing can be reasonably called one of the grandfathers of AI, along with early pioneers such as Vannevar Bush.

That brings me to the topic of this talk, "To dream the possible dream." AI is thought to be an impossible dream by many. But not to us in AI. It is not only a possible dream, but, from one point of view, AI has been a reality that has been demonstrating results for nearly 40 years. And the future promises to generate an impact greater by orders of magnitude than progress to date. In this talk I will attempt to demystify the process of what AI researchers do and explore the nature of AI and its relationship to algorithms and software systems research. I will discuss what AI has been able to accomplish to date and its impact on society. I will also conclude with a few comments on the long-term grand challenges.

Human and Other Forms of Intelligence

Can a computer exhibit real intelligence? Simon provides an incisive answer: "I know of only one operational meaning for 'intelligence.' A (mental) act or series of acts is intelligent if it accomplishes something that, if accomplished by a human being, would be called intelligent. I know my friend is intelligent because he plays pretty good chess (can keep a car on the road, can diagnose symptoms of a disease, can solve the problem of the missionaries and cannibals, etc.) I know that computer A is intelligent because it can play excellent chess (better than all but about 200 humans in the entire world). I know that Navlab is intelligent because it can stay on the road. The trouble with those people who think that computer intelligence is in the future is that they have never done serious research on human intelligence. Shall we write a book on 'What Humans Can't Do?' It will be at least as long as Dreyfus' book. Computer intelligence has been a

Can AI equal human intelligence? Some philosophers and physicists have made successful lifetime careers out of attempting to answer this question. The answer is, AI can be both more and less than human intelligence.

fact at least since 1956, when the Logic Theory machine found a proof that was better than the one found by Whitehead and Russell, or when the engineers at Westinghouse wrote a program that designed electric motors automatically. Let's stop using the future tense when talking about computer intelligence."

Can AI equal human intelligence? Some philosophers and physicists have made successful lifetime careers out of attempting to answer this question. The answer is, AI can be both more and less than human intelligence. It doesn't take large tomes to prove that they cannot be 100% equivalent. There will be properties of human intelligence that may not be exhibited in an AI system (sometimes because we have no particular reason for doing so or because we have not yet gotten around to it). Conversely, there will be capabilities of an AI system that will be beyond the reach of human intelligence. Ultimately, what will be accomplished by AI will depend more on what society needs and where AI may have a *comparative advantage* than on philosophical considerations.

Let me illustrate the point by two analogies that are not AI problems in themselves but whose solutions require some infusion of AI techniques. These problems, currently at the top of the research agenda within the information industry, are *digital libraries* and *electronic commerce*.

The basic unit of a digital library is an electronic book. An electronic book provides the same information as a real book. One can read and use the information just as we can in a real book. However, it is difficult to lie in bed and read an electronic book. With expected technological advances, it is conceivable a subnotebook computer will weigh less than 12 ounces and have a 6" x 8" high resolution color screen, making it look and feel like a book that you might read in bed. However, the analogy stops there. An electronic book cannot be used as part of your rare book collection, nor can it be used to light a fire on a cold night to keep you warm. You can probably throw it at someone, but it would be expensive. On the other hand, using an electronic book, you can process, index, and search for information; open the right page; highlight information; change font size if you don't

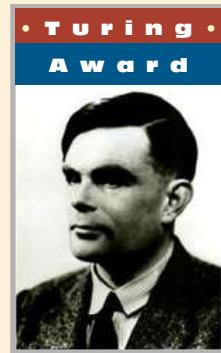
have your reading glasses; and so on. The point is, an electronic book is not the same as a real book. It is both more and less.

A key component of electronic commerce is the electronic shopping mall. In this virtual mall, you can walk into a store, try on some *virtual* clothing, admire the way you look, place an order and have the real thing delivered to your home in 24 hours. Obviously, this does not give you the thrill of going into a real mall, rubbing shoulders with real people and trying on real clothing before you make your purchase. However, it also eliminates the problems of getting dressed, fighting the traffic and waiting in line. More importantly, you can purchase your dress in Paris, your shoes in Milan and your Rolex in Hong Kong without ever leaving your home. Again, the point is that an electronic shopping mall is not the same as a real shopping mall. It is both more and less.

Similarly, AI is both more and less than human intelligence. There will be certain human capabilities that might be impossible for an AI system to reach. The boundary of what can or cannot be done will continue to change with time. More important, however, it is clear that some AI systems will have super human capabilities that would extend the reach and functionality of individuals and communities. Those who possess these tools will make the rest of us look like primitive tribes. By the way, this has been true of every artifact created by the human species, such as the airplane. It just so happens that AI is about creating artifacts that enhance the mental capabilities of the human being.

AI and Algorithms

Isn't AI just a special class of algorithms? In a sense it is; albeit a very rich class of algorithms, which have not yet received the attention they deserve. Second, a major part of AI research is concerned with problem definition rather than just problem solution. Like complexity theorists, AI researchers also tend to be concerned with NP-complete problems. But unlike their interest in the complexity of a given problem, the focus of research in AI tends to revolve around finding algorithms that provide approximate, satisfying solutions with no guarantee of optimality.



The concept of satisfying solutions comes from Simon's pioneering research on decision making in organizations leading to his Nobel Prize. Prior to Simon's work on human decision making, it was assumed that, given all the relevant facts, the human being is capable of rational choice weighing all the facts. Simon's research showed that "computational constraints on human thinking" lead people to be satisfied with "good enough" solutions rather than attempting to find rational optimal solutions weighing all the facts. Simon calls this the principle of "bounded rationality." When people have to make decisions under conditions which overload human thinking capabilities, they don't give up, saying the problem is NP-complete. They use strategies and tactics of *optimal-least-computation search* and not those of *optimal-shortest-path search*.

Optimal-least-computation search is the study of approximate algorithms that can find the best possible solution given certain constraints on the computation, such as limited memory capacity, limited time, or limited bandwidth. This is an area worthy of serious research by future complexity theorists!

Besides finding solutions to exponential problems, AI algorithms often have to satisfy one or more of the following constraints: exhibit adaptive goal-oriented behavior, learn from experience, use vast amounts of knowledge, tolerate error and ambiguity in communication, interact with humans using language and speech, and respond in real time.

Algorithms that exhibit adaptive goal oriented behavior. Goals and subgoals arise naturally in problems where algorithm specification is in the form of "What" is to be done rather than "How" it is to be done. For example, consider the simple task of asking an agent, "Get me Ken." This requires converting this goal into subgoals, such as look up the phone directory, dial the number, talk to the answering agent, and so on. Each subgoal must then be converted from What to How and executed. Creation and execution of plans has been studied extensively within AI. Other systems such as report generators, 4GL systems, and data base query-by-example methods, use simple template-based solutions to solve the "What to

How" problem. In general, to solve such problems, an algorithm must be capable of creating for itself an agenda of goals to be satisfied using known operations and methods, the so-called "GOMs approach." Means-ends analysis, a form of goal-oriented behavior, is used in most expert systems.

Algorithms that learn from experience. Learning from experience implies the algorithm has built-in mechanisms for modifying internal structure and function. For example, in the "Get me Ken" task, suppose Ken is ambiguous and you help the agent to call the right Ken; next time you ask the agent to "Get me Ken," it should use the same heuristic that you used to resolve the ambiguity. This implies the agent is capable of acquiring, representing, and using new knowledge and engaging in a clarification dialog, where necessary, in the learning process. Dynamic modification of internal structure and function is considered to be dangerous in some computer science circles because of the potential for accidental overwriting of other structures. Modifying probabilities and modifying contents of tables (or data structures) have been used successfully in learning tasks where the problem structure permits such a formulation of learning. The Soar architecture developed by Newell et al., which uses rule-based system architecture, is able to discover and add new rules (actually "productions") and is perhaps the most ambitious undertaking to date to create a program that improves with experience.

Algorithms that interact with humans using language and speech. Algorithms that can effectively use speech and language in human-computer interface will be essential as we move toward a society where nonexperts use computers in their day-to-day problems. In the previous example of the "Get me Ken" task, unless the agent can conduct the clarification dialog with the human master using language and speech or some other natural form of communication, such as "Form Filling," widespread use of agents will be a long time coming. Use of language and speech involves creating algorithms that can deal with not only ambiguity and nongrammatical-

*It just so happens that AI is about creating artifacts that enhance
the mental capabilities of the human being.*

ity but also with parsing and interpreting of natural language with a large, dynamic vocabulary.

Algorithms that can effectively use vast amounts of knowledge. Large amounts of knowledge not only require large memory capacity but creates the more difficult problem of selecting the right knowledge to apply for a given task and context. There is an illustration that John McCarthy is fond of using. Suppose one asks the question, "Is Reagan sitting or standing right now?" A system with a large database of facts might proceed to systematically search the terabytes of data before finally coming to the conclusion that it does not know the answer. A person faced with the same problem would immediately say, "I don't know," and might even say, "and I don't care." The question of designing algorithms "that know what they do not know" is currently an unsolved problem. With the prospect of very large knowledge bases looming around the corner, "knowledge search" will become an important algorithm design problem.

Algorithms that tolerate error and ambiguity in communication. Error and ambiguity are a way of life in human-to-human communication. Warren Teitelman developed nearly 25 years ago an interface called "Do What I Mean (DWIM)." Given the requirements of efficiency and getting the software done in time, all such ideas were deemed to be frivolous. Today, with the prospect of Giga PCs around the corner, we need to revisit such error-forgiving concepts and algorithms. Rather than saying "illegal syntax," we need to develop algorithms that can detect ambiguity (i.e., multiple possible interpretations, including null) and resolve it where possible by simultaneous parallel evaluation or by engaging in clarification dialog.

Algorithms that have real-time constraints. Many software systems already cope with this problem, but only through careful, painstaking analysis of the code. Not much work has been done on how to create algorithms that can accept a "hurry-up" command! One approach to this problem appears in the Prodigy system, which generates an approximate plan immediately and gradually improves

and replaces that plan with a better one as the system finds more time to think. Thus, it always has an answer but the quality of the answer, improves with time. It is interesting to note that many of the iterative algorithms in numerical analysis can be recast in this form of "anytime answers."

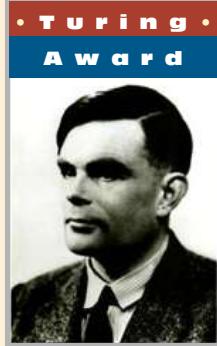
Algorithms with self-awareness. Algorithms that can explain their capabilities (e.g., a Help command capable of answering "how-to" and "what-if" questions), and monitor, diagnose, and repair themselves in the presence of viruses require internal mechanisms that can be loosely called "self-awareness." Online hypertext manuals and checksums are simpler examples of such mechanisms. Answering how-to and what-if questions is harder. Monitoring and diagnosis of invading viruses implies interpreting incoming action requests rather than just executing them.

Such algorithm design issues seem to arise naturally in AI, and numerous solutions have been created by AI researchers in specific contexts. Further development and generalization of such solutions will enrich all of computer science.

Software Systems and AI

Isn't AI just software? Isn't a TV set just electronics? In a general sense, AI is just software, although AI systems tend to get large and complex. Attempting to build AI systems often implies building the necessary software tools. For example, AI researchers were responsible for many of the early advances in programming languages, such as list structures, pointers, virtual memory, dynamic memory allocation, and garbage collection, among others.

Large AI systems, especially those that are deployed and in daily use, share many of the common problems that are observed in developing other complex software systems, i.e., the problems of "not on time," "over budget," and "brittle." The "mythical man-month" principle affects AI systems with a vengeance. Not only do these systems tend to be large and complex, but they often cannot fall back on a learning curve based on having designed similar systems before. Thus, it is difficult to formulate requirement specifications or testing pro-



cedures as is usual in conventional software engineering tasks.

There are a number of new concepts that are emerging within the context of AI that suggest new approaches and methodologies for all of computer science:

- *Plug-and-play architectures.* To produce an integrated concept demonstration of an AI system that routinely uses components that learn from experience, use knowledge, tolerate error, use language, and operate in real time, one cannot start from scratch each time. The application component of such systems can be less than 10% of the total effort. If the system is to be operational in a reasonable amount of time, one needs to rely on interfaces and software architectures that consist of components that plug and play together. This idea is similar to the old blackboard concept: cooperating agents that can work together but don't require each other explicitly.
- *The 80/20 rule.* The last 40 years of attempting to build systems that exhibit intelligent behavior has shown us how difficult the task really is. A recent paradigm shift is to move away from autonomous systems that attempt to entirely replace a human capability to systems that support a human master as intelligent agents. For example, rather than trying to create a machine translation system, create a *translation assistant* that provides the best alternative interpretations to a human translator. Thus, the goal would be to create a system that does 80% of a task and leaves the remaining 20% to the user. Once such a system is operational, the research agenda would target the remaining 20% and repeat the 80/20 rule on the next version. Thus, the system would incrementally approach human performance, while at all times providing a usable artifact that improves human productivity by factors of 3 to 5 after every iteration. However, the task is not as simple as it may appear. With this new paradigm, the old problem of "how can a system know what it does not know" raises its ugly head. For an intelligent agent to be able to say, "Please wait, I will call my supervisor," it must be self-aware! It must know what it can and cannot do. Not an easy task either, but one that needs to be

solved anyway.

- *Fail-fast strategies.* In engineering design there are two accepted practices: "get it right the first time" and "fail fast." The former is used when one is designing a product that has been produced many times before. In building systems or robots that have never been built before, attempting to "get it right the first time" may not be the best strategy. Our recent experiment for NASA in building the Dante robot for exploring volcanic environments is one such example. Most NASA experiments are required to be fail-safe, because in many missions human lives are at risk. This requirement to be error free leads to 15-year planning cycles and billion-dollar budgets. The fail-fast strategy says that if you are trying to build a complex system that has never been built before, it is prudent to build a series of throw-away systems and improve the learning curve. Since such systems will fail in unforeseeable ways, rather than viewing failure as an unacceptable outcome one should view failure as a stepping-stone to success. Our first Dante failed after taking a few steps. The second one lasted a week. We are now working on a third generation model. Both time and cost expended on this experiment is at least an order of magnitude smaller than those for comparable conventional missions.
- *The scientific method.* At an NRC study group meeting, a physicist asked, "Where is the science in computer science?" I am happy to say that we are beginning to have examples of the "hypothesis, experiment, validation, and replication" paradigm within some AI systems. Much of the speech-recognition research has been following this path for the past 10 years. Using common databases, competing models are evaluated within operational systems. The successful ideas then seem to appear magically in other systems within a few months, leading to validation or refutation of specific mechanisms for modeling speech. ARPA deserves a lot of the credit for requiring the community to use such validation procedures. All of experimental computer science could benefit from such disciplined experiments. As Newell used to say, "Rather than

**AI continues to be a possible dream worthy of dreaming.
Advances in AI have been significant. AI will continue to be
the generator of interesting problems and solutions.**

argue, let us design an experiment to prove or disprove the idea!"

AI and Society

Why should society support AI research and more broadly, computer science research? The information technology industry, which was nonexistent 50 years ago, has grown to be over 10% of the GNP and is responsible for over 5% of the total employment in the country. While AI has not played a substantial role in the growth of this industry, except for the expert system technology, it appears possible that we are poised on the threshold of a number of important applications that could have a major impact on society. Two such applications include an accident-avoiding car and a reading coach:

The Navlab: The Carnegie Mellon Navlab project brings together computer vision, advanced sensors, high-speed processors, planning, and control to build robot vehicles that drive themselves on roads and cross country.

The project began in 1984 as part of ARPA's Autonomous Land Vehicle (ALV) program. In the early 1980s, most robots were small, slow indoor vehicles tethered to big computers. The Stanford Cart took 15 minutes to map obstacles, plan a path, and move each meter. The CMU Imp and Neptune improved on the Cart's top speed, but still moved in short bursts separated by long periods of looking and thinking. In contrast, ARPA's 10-year goals for the ALV were to achieve 80 kph on roads and to travel long distances across open terrain.

The Navlab, built in 1986, was our first self-contained testbed. It had room for on-board generators, on-board sensors, on-board computers, and, most importantly, on-board graduate students. Researchers riding on board were in a much better position to observe its behavior and to debug or modify the programs. They were also highly motivated to get things working correctly.

The latest is the Navlab II, an army ambulance HMMWV. It has many of the sensors used on earlier vehicles, plus cameras on pan/tilt mounts and three aligned cameras for trinocular stereo vision. The HMMWV has high ground clearance for dri-

ving on rough terrain, and a 110-kph top speed for highway driving. Computer-controlled motors turn the steering wheel and control the brake and throttle.

Perception and planning capabilities have evolved with the vehicles. ALVINN is the current main road-following vision system. ALVINN is a neural network, which learns to drive by watching a human driver. ALVINN has driven as far as 140 km, and at speeds over 110 kph.

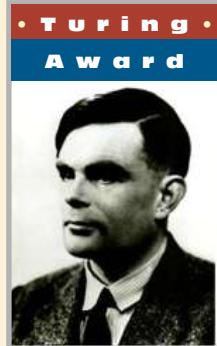
Ranger finds paths through rugged terrain. It takes range images, projects them onto the terrain, and builds Cartesian elevation maps. Ranger has driven the vehicle for 16 km on our test course.



SMARTY and D* find and follow cross-country routes. D* plans a route using A* search. As the vehicle drives, SMARTY finds obstacles using GANESHA's map, steers the vehicle around them and passes the obstacles to D*. D* adds the new obstacles to its global map and replans the optimal path.

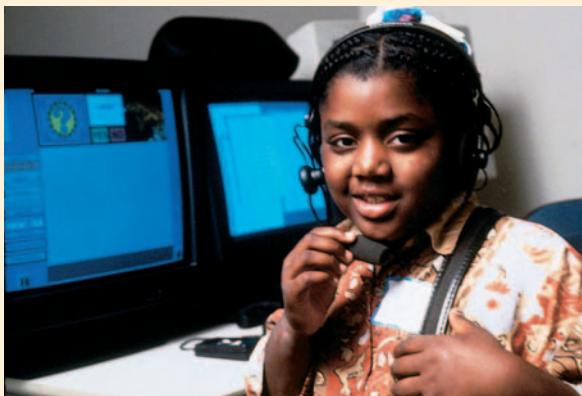
Other Navlab modules include RACCOON, which follows a lead vehicle by tail-light tracking; YARF, which tracks lines and detects intersections; and PANACEA, which controls a pan/tilt mount to see around curves.

The 10-year goals of the ALV program have been met. Navlab technology is being integrated with specialized planners and sensors to demonstrate practical missions, and a new project funded by the Department of Transportation is investigating Navlab technology for preventing highway accidents. As basic driving capabilities mature, the Navlab continues to provide new opportunities



both for applications and for continued research in perception, planning, and intelligent robot systems.

There are six million automotive accidents in the U.S. each year, costing over \$50 billion to repair. These accidents result in 40,000 fatalities, 96% of which are caused by driver error. Using Navlab technology can eliminate a significant fraction of those accidents, given appropriate sensors and computation integrated into each car. AI and computer science researchers can justifiably be proud of such a contribution to society.



The LISTEN Project: At Carnegie Mellon, Project LISTEN is taking a novel approach to the problem of illiteracy. We have developed a prototype automated reading coach that listens to a child read aloud, and helps when needed. The system is based on the CMU Sphinx II speech recognition technology. The coach provides a combination of reading and listening, in which the child reads wherever possible and the coach helps wherever necessary—a bit like training wheels on a bicycle.

The coach is designed to emphasize comprehension and ignore minor mistakes, such as false starts or repeated words. When the reader gets stuck, the coach jumps in, enabling the reader to complete the sentence. When the reader misses an important word, the coach rereads the words that led up to it, just like the expert reading teachers after whom the coach is modeled. This context often helps the reader correct the word on the second try. When the reader runs into more difficulty,

the coach rereads the sentence to help the reader comprehend it. The coach's ability to listen enables it to detect when and where the reader needs help. Further research is needed to turn this prototype into robust educational software. Experiments to date suggest that it has the potential to reduce children's reading mistakes by a factor of five and enable them to comprehend material at least six months more advanced than they can read on their own.

Illiteracy costs the U.S. over \$225 billion dollars annually in corporate retraining, industrial accidents, and lost competitiveness. If we can reduce illiteracy by just 20%, Project LISTEN could save the nation over \$45 billion a year.

Other AI research to have a major impact on society includes Feigenbaum's early research on knowledge systems; Kurzweil's work on reading machines for the blind; Papert, Simon, Anderson, and Shank's contributions to learning by doing; and the robotics work at MIT, Stanford, and CMU.

Several new areas where advances in AI are likely to have an impact on society are:

- Creation of intelligent agents to monitor and manage the information glut by filtering, digesting, abstracting, and acting as an agent of a human master;
- Making computers easier to use: use of multi-modal interfaces, DWIM techniques, intelligent help, advice giving agents, and cognitive models;
- Aids for the disabled: Development of aids for people with sensory, cognitive, or motor disabilities based on AI technologies appears promising, especially the area of creating aids for cognitive disabilities such as memory and reasoning deficiencies; and
- Discovery techniques: Deriving knowledge and information for a large amount of data, whether the data is scientific or financial, has the potential for a major impact.

Whether it is to save lives, improve education, improve productivity, overcome disabilities, or foster discovery, AI has produced and will continue to produce research results with the potential to have



a profound impact on the way we live and work in the future.

Grand Challenges in AI

What's next for AI? There are several seemingly reasonable problems which are exciting and challenging, and yet are currently unsolvable. Solutions to these problems will require major new insights and fundamental advances in computer science and artificial intelligence. Such challenges include: a world champion chess machine, a translating telephone, and discovery of a major mathematical result by a computer, among others. Here, I will present two such grand challenges which, if successful, can be expected to have a major impact on society:

Self-Organizing Systems: There has been a long and continuing interest in systems that learn and discover from examples, from observations, and from books. Currently, there is a lot of interest in neural networks that can learn from signals and symbols through an evolutionary process. Two long-term grand challenges for systems that acquire capability through development are: read a chapter in a college freshman text (say, physics or accounting) and answer the questions at the end of the chapter; and learn to assemble an appliance (such as a food processor) from observing a person doing the same task. Both are extremely hard problems, requiring advances in vision, language, problem-solving techniques, and learning theory. Both are essential to the demonstration of a self-organizing system that acquires capability through (possibly unsupervised) development.

Self-Replicating Systems: There have been several theoretical studies in this area since the 1950's. The problem is of some practical interest in areas such as space manufacturing. Rather than uplifting a whole factory, is it possible to have a small set of machine tools that can produce, say, 80% of the parts needed for the factory (using the 80/20 rule discussed earlier), using locally available raw materials and assemble it in situ? The solution to this problem of manufacturing on Mars involves many

different disciplines, including materials and energy technologies. Research problems in AI include knowledge capture for replication; design for manufacturability; and design of systems with self-monitoring; self diagnosis; and self-repair capabilities.

Conclusion

Let me conclude by saying that AI continues to be a possible dream worthy of dreaming. Advances in AI have been significant. AI will continue to be the generator of interesting problems and solutions. However, ultimately the goals of AI will be realized only with developments within the broader context of computer science, such as the availability of a Giga-PC—i.e., a billion-operations-per-second computer at PC prices—and advances in software tools, environments, and algorithms.

Acknowledgment

I was fortunate to have been supported by ARPA for nearly 30 years. I would like thank Lick Licklider, Ivan Sutherland, Larry Roberts, and Bob Kahn for their vision in fostering AI research even though it was not part of their own personal research agenda, and the current leadership at ARPA: Duane Adams, Ed Thompson, and Allen Sears, for continuing the tradition.

I am grateful to Herb Simon, Ed Feigenbaum, Jaime Carbonell, Tom Mitchell, Jack Mostow, Chuck Thorpe, and Jim Kocher for suggestions, comments, and help in the preparation of this lecture.

RAJ REDDY, Herbert A. Simon University Professor, is Dean of the School of Computer Science at CMU. His mailing address is Carnegie Mellon University, School of Computer Science, 5325 Wean Hall, Pittsburgh, PA 15123-3890. email:rr@cmu.edu.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

Improving the Future by Examining the Past

Charles P. Thacker
Microsoft Research

Abstract: During the last fifty years, the technology underlying computer systems has improved dramatically. As technology has evolved, designers have made a series of choices in the way it was applied in computers. In some cases, decisions that were made in the twentieth century make less sense in the twenty-first. Conversely, paths not taken might now be more attractive given the state of technology today, particularly in light of the limits the field is facing, such as the increasing gap between processor speed and storage access times and the difficulty of cooling today's computers.

In this talk, I'll discuss some of these choices and suggest some possible changes that might make computing better in the twenty-first century.

Categories & Subject Descriptors: C.0 Computer Systems Organization; General.

General Term: Design.

THE SEARCH FOR PERFORMANCE IN SCIENTIFIC PROCESSORS

JOHN COCKE

I am honored and grateful to have been selected to join the ranks of ACM Turing Award winners. I probably have spent too much of my life thinking about computers, but I do not regret it a bit. I was fortunate to enter the field of computing in its infancy and participate in its explosive growth. The rapid evolution of the underlying technologies in the past 30 years has not only provided an exciting environment, but has also presented a constant stream of intellectual challenges to those of us trying to harness this power and squeeze it to the last ounce. I hasten to say, especially to the younger members of the audience, there is no end in sight. As a matter of fact, I believe the next thirty years will be even more exciting and rich with challenges.

The three principal contributors to performance in scientific processors are the algorithm, the compiler, and the machine organization. When possible, the simultaneous optimization of these three factors holds the key to the highest possible performance. Of the three contributors, algorithm improvements are the most important. An idea that changes an algorithm from N^{**2} to $N^* \log N$ operations, where N is proportionate to the number of input elements, is considerably more spectacular than an improvement in machine organization, where only a constant factor of run-time is achieved. Unfortunately, really exploiting the characteristics of a particular algorithm in the underlying machine organization often results in a specialized computer that will not perform well on most other algorithms. Signal processors are an example of this. They handle information transforms very fast, even in one clock cycle, but can be extremely inefficient in computing anything else.

My interests have been in achieving high performance for a broad range of scientific calculations. There-

fore, I have focused mainly on optimizations involving the compiler and the underlying machine architecture. It is very probable that in the future, the highest possible performance improvements must also include the more difficult problem of algorithm modification. This is because parallelism will be essential, and the performance of parallel machines depends heavily on the algorithm used and how well it matches the parallel architecture.

In the last 25 years we have seen a 100-fold increase in uniprocessor computer performance. We have also seen a 10,000-fold decrease in cost. In the next 25 years, we are likely to see similar cost improvements. However, because subnanosecond cycles will be difficult, the performance of our largest machines will be on the order of a few billion instructions per second. We should always look at the absolute performance of uniprocessors, because if we can partition the problem, multiple processors may be used to gain even higher performance. We have seen this concept in trivial use for years, for example in job entry scheduling programs. Therefore, a key unresolved question is to understand how to partition a given problem at a global level and then operate on its separate parts using parallel machines. Global partitioning is a very difficult problem. Today there are many ideas, some good at specialized tasks, yet there is nothing promising for a uniform approach.

Today's compiler technology is quite sophisticated. Those who build machine architectures and organizations should design them so the known compiler optimization techniques are easily applied. New compiler concepts will be needed to exploit the capabilities of the new machines. I also believe that additional improvements are needed and are possible for existing optimizations. Better register allocation is one example.

The construction of a special purpose processor of

very high performance is not difficult compared to a general purpose processor. By "special purpose," I mean concepts such as floating point units, caches, and vector processors. A machine built with a cache and a vector unit is specialized toward problems that have a good cache-hit ratio and are amenable to vectorization. It will have poor cost/performance on other problems. With relatively little progress on the partitioning problem and an apparent limit on the speed of a uniprocessor, we may find tomorrow's high performance world not much different than today's. Indeed, there will be a large class of problems that cannot be partitioned practically and thus, will be limited to uniprocessor performance.

I would like to describe to you the three most interesting projects that I have participated in. They were interesting because I felt I learned faster during these projects than at other times.

When I joined IBM in 1956, the Stretch project was underway. The project was led by Steve Dunwell, who placed performance, not cost, as paramount. My experience had consisted of writing one Monte Carlo simulation where the machine language was HEX. I am shocked that I did not write an assembler before proceeding, but the problem was so interesting that such a thing never occurred to me.

Designed for Los Alamos, Stretch's ambitious goal was to be 100 times faster than the existing 704 while providing great flexibility in addressing, floating-point arithmetic, and nonnumeric operations. Any bit in the machine could be addressed directly, any word could be monitored, variable length data could be referenced, and floating point numbers came in many varieties. It was a programmer's dream—especially assembly language level programmers—and it was a wonderful challenge for those of us designing the hardware.

To overlap memory access, we introduced instruction execution look ahead (pipelining). Error Correcting Code (ECC), was applied to main memory and disk and tape I/O, and we worked with the compiler people to develop efficient instruction sequencing and register allocation.

Fortran I for the 704 generated excellent code, even measured by today's criteria. Those interested in compilers learned a great deal from Fortran's usage with Stretch. For example, much of the richness of the Stretch architecture was not exploited by the compiler. As I recall, John Backus told us in advance that this would be the case. We architects would have benefited had we known more about the Fortran compiler at the time. Because of this and many other experiences, however, we have learned that it is easier to write compilers that capitalize on a simpler instruction set. Although Stretch met less than half of its performance goal when it was shipped to Los Alamos in 1961, we had invented and tried many techniques still used today.

Next, I would like to tell you a little bit about the Advanced Computer System (ACS). This was a project we undertook between 1964 and 1968. It had a simple

yet irresistible goal: to design and build the fastest scientific computer feasible. Under the late Jack Bertram's leadership, we designed a computer with many organizational features that even today are not well known.

For various reasons, however, ACS was not built. It would have had a ten nanosecond cycle time, on the order of today's multimillion dollar computers, so it would have been very fast hardware. This is only where machine organization begins. At each cycle we dispatched seven operations, one to the branch unit, three to the fixed point unit, and three to the floating point unit. The fixed point unit could initiate three instructions on each cycle. The floating point unit had a buffer of eight operands and logic to pick the first three out of the eight that were ready. We had two paths to cache allowing two memory accesses per cycle, and to match the cache access time to the CPU cycle, it was pipelined five deep (necessary because of the low level of circuit integration). A neat invention was a FIFO store queue that allowed computation to proceed without waiting for a store-through and a hardware interlock to protect the cases when a reference is made to the operand. It also resolved interfering load instructions.

Much of our effort went into dealing with branch instructions in ways that minimized draining the pipeline. We divided each branching instruction into its three essential operations: determination of condition, calculation of branch target address, and the actual jump instruction. There was also a branch history table that allowed instruction prefetching based on the dynamics of recent branch instruction executions. Another specialized form of branch was called "skip;" it allowed compiler scheduling across branches. Given all of these hardware assists, the experimental compiler generated good code and scheduled instructions to minimize the pipeline effects.

Long before we had a firm hardware design, we had an experimental optimizing compiler to evaluate variants on the design. Working with a combined team of compiler and hardware people, I learned the importance of not including hardware features that the compiler could not use and including hardware facilities to allow efficient compilation. This experimental compiler was the source for much of our subsequent work on optimization algorithms. Many of the code optimization methods used in compilers today came from this work. These include interval-based control flow analysis, data flow analysis, common subexpression elimination, code motion, strength reduction, and code scheduling. On occasion, the compiler was able to generate better code for ACS than the best hand coders. We learned of the importance of an efficient compiler and its surrounding software support tools.

The parts of ACS that were built 20 years ago are still impressive, except for the speed-power product: internal circuits were 250 picoseconds, each circuit dissipated about 30 milliwatts, and there were up to 40 circuits per chip. Had the complete machine been built, I believe that the cycle time would have been in the

low teens of nanoseconds. It would have accomplished four to five instructions per cycle on linear algebra-type problems. But, because of cache latency and the inability to almost always anticipate the correct branch flow, we would not have achieved near that rate on more general purpose problems.

Another important part of the 801 project was the tight coupling and simultaneous development of the hardware and the compiler.

These are only a few of the features we had in ACS. Many ideas have found their way into subsequent machines at IBM, particularly in the cache area. Some of the more intricate ones, however, have not yet been used. In many cases this was due to cost versus performance trade-offs, but I am confident that declining costs will enable use of these ideas.

ACS never made it out of the laboratory; I suppose it was too big and too expensive, but for me it was probably the most exciting project I have ever been involved in. In reflecting on this, I believe that what made it particularly exciting was that we were a small team, mostly hand-picked by Jack Bertram, and we pioneered every aspect of the project.

I spoke about the architecture and the compiler, but some of our most clever inventions had to do with testing instruments and techniques, fast disks, packaging, and cooling. I received a great deal of satisfaction in watching our engineers tackle and solve these problems. Another principal factor was the quality of the team. Many are names I am sure you will recognize—among them Fran Allen, Dick Arnold, Fred Buelow, Phil Dauber, John Earle, Charlie Freiman, Russ Robelen, Herb Schorr, and Ed Sussenguth. My only regret about ACS is that unlike compiler ideas, we did not take the time to publish our ideas on hardware so others could build on them.

Let me now move on to discuss another interesting project I was involved in—the 801 computer. Around 1974, we were investigating the possibility of building an all digital telephone exchange capable of handling approximately one million calls per hour. At approximately 20,000 instructions per call setup, we calculated that we needed a processor capable of executing 6 million instructions per second (MIPS). But since we did not know that much about telephones, we felt that we should probably aim for a 12 MIPS processor. Certainly, a general purpose machine was not the best choice for this task. For example, we had stringent real-time response requirements. So in a sense, we were looking at a special purpose machine. From the beginning, we also assumed that we would program in a high-level language. In this case it was not absolute performance that motivated us. We had a performance target. We knew the general nature of the applications; we had no heavy floating point calculations, and we were looking for the

lowest cost computer that would meet these requirements. We designed a machine that subsequently became known as the 801 Computer. Eventually, we abandoned the idea of a telephone exchange, and concluded that we had pretty powerful ideas in computer design. We complemented these ideas with a high level

language and a compiler, PL.8, and pursued their development for the next several years.

Among the principal features of the 801 were separate instruction and data caches, providing much higher bandwidth between the memory and CPU, and no arithmetic operations to storage, thereby greatly simplifying pipelining. Another important part of the 801 project was the tight coupling and simultaneous development of the hardware and the compiler. This tight coupling allowed the construction of an effective compiler and provided a simple machine organization.

This project was a great team effort, and my colleagues Marc Auslander, Greg Chaitin, Al Chang, Marty Hopkins, Peter Markstein, and George Radin, to name just a few, were not only great contributors, but made the whole venture enjoyable. Another satisfying aspect of the 801 project was that many of our ideas were considered sufficiently interesting by others and stimulated considerable additional research and experimentation in many universities. Berkeley coined the name "RISC" (Reduced Instruction Set Computing) for similar work.

Within IBM we feel it is extremely important to simulate the hardware of a new machine extensively. As the complexity of machines increased, the time required to simulate them was becoming too long. In 1980 we started to develop a special purpose machine for this task. Rick Malm developed an early prototype and Monty Denneau designed, built, and debugged a very large system. Its goal was to provide a hardware assist to the logic design simulation, increasing the speed of simulation 100-fold to 1000-fold, depending on the number of circuits to be simulated. Logic simulation appears to be an ideal problem for parallel machines because it should simply emulate the way a computer really works. The machine has 256 parallel logic units accessing a shared memory. A special compiler takes a reasonably high-level logic description, generates the necessary instructions, and loads them into the machine. Each unit simulates about 4000 circuits sequentially and at the end of each cycle, broadcasts its results through a nonblocking switch to each of its 255 partners. Thus, 256 circuits per cycle are simulated.

Assigning logic blocks to a machine is analogous to assigning circuits to chips, and according to Rent's Law, 4000 I/Os for 4000 circuits should be more than

enough. In spite of this, the compiler partitioning problem takes a very long time on a large mainframe CPU. This forces us to use the machines only for long-running simulations. In spite of this time-consuming difficulty, many such machines are in constant use at IBM. The next generation logic simulator, with added switching capacity, is being built to ease this problem.

Note, however, that the nonblocking switches grow at least as fast as $N^* \log N$, whereas the machine hardware grows linearly. Thus, in the limit, it will be *all-switch*. I have mentioned this example because it demonstrates a paradox simply: A priori, it seems very natural and simple to map a logic simulator onto multiple machines, but a simplistic straightforward extension will encounter difficulties.

processors. For example, on an inner loop that goes at three instructions per cycle on a machine that has a cache taking 10 cycles per miss, a 3 percent miss ratio will halve performance.

Thus, machines such as this can be considered specialized in the sense that problems with good cache hit ratios achieve high performance. I see many techniques where tricky programming can vastly improve the cache hit ratio. I believe that people interested in compilers should try to make these tricky techniques automatic, as vectorization has been made automatic by compilers.

I mentioned earlier that improved algorithms can provide the most leverage, orders of magnitude in the dimension of the problem, whereas parallelism will at

It is this trend that emphasizes cost over performance that leads me to believe the search for future scientific computing performance has to concentrate on gross parallelism.

This brings me to the future. By exploiting improvements in circuit and memory density (e.g., one million transistor chips and four megabit chips), it is possible with just a few CMOS chips to build a powerful scientific machine that will match the performance of vector based computers on many important problems. This machine will have a very fast floating point multiply/add unit equipped with buffers on a single chip. It also will be capable of simultaneously executing a fixed point, a floating point, and a branch instruction. As has been stated several times, optimizing compilers will be essential to exploit such machines.

Memory is inexpensive and cost improvements will continue. Thus, we can expect large random-access memories consisting of hundreds of gigabytes. Disks will maintain the backup database, but by paging, the database applications will execute out of main memory at speeds substantially faster than they do today. The significance of this is that the enormous amount of effort that has gone into developing schemes to achieve high performance on mechanical devices will no longer be necessary.

As I said before, I expect the thirty-year-old trend of 100-fold computer performance improvement with 10,000-fold cost improvement to continue. It is this trend, that emphasizes cost over performance, that leads me to believe the search for future scientific computing performance has to concentrate on gross parallelism. It is necessary due to increasing difficulties in reducing cycle time and cycles per instruction. Due to the continuing steep decrease in costs, it will be possible to aggregate many CPUs executing multiple instruction streams concurrently. Thus, one of the principal challenges will lie in compiler optimization techniques, both to recognize the parallelism and schedule the instructions. One particularly difficult yet crucial problem is to minimize cache misses in these parallel pro-

grams. New algorithms tailored to such parallel machines that exploit the clever details of their architecture and interconnection will be essential, and conversely, specialized machines oriented towards such parallel algorithms will be possible and economically warranted.

Many years ago, John McCarthy remarked to me that if it had not been for Alan Turing and his idea of a Universal Machine, we would still be arguing about the capabilities of different machine designs. Fortunately, this is not an issue, however, we do seem to carry on at length about cost/performance of various machine designs. The arguments are further complicated since each machine can demonstrate a set of problems and algorithms for which it is *specialized* and does particularly well. I am certain this will be worse for parallel machines where the degree of specialization can be higher. Comparison of various machine architectures is not done well today, perhaps because of its intrinsic difficulty or perhaps because of commercial implications. I hope this can be done more carefully in the future; maybe we could look to academia for help.

I would like to conclude by reiterating that the past thirty years have emphasized cost improvement over performance. I see no reason why this trend should not continue. As we look ahead, we see ourselves approaching a limit on the performance of a uniprocessor and therefore, we observe many working on multiple machine aggregates. As it is not obvious that this will solve the high performance scientific computing problem, we should not infer that the future is simply an extrapolation of today's ideas. I do not find it discouraging that there seems to be no clear-cut route to high performance. I feel the flexibility of computers will allow us to solve problems in ways not yet envisioned, and will make the future of computing more interesting than the past.

COMPUTER SCIENCE: THE EMERGENCE OF A DISCIPLINE

The continued rapid development of computer science will require an expansion of the science base and an influx of talented new researchers. Computers have already altered the way we think and live; now they will begin to elevate our knowledge of the world.

JOHN E. HOPCROFT

It is a great honor to be a recipient of the 1986 Turing Award. Along with the recognition provided by the award comes the opportunity to present this lecture, and so to speak not only to computer scientists, but also to policy makers and to other members of the scientific establishment. I would like to start by relating some of my experiences in the field of computer science, and then to make some recommendations about the future development of the discipline.

I began my professional career in computer science in 1964, when computer science was just beginning to establish itself as an academic discipline. Having been a part of the academic community during the formative years of computer science, I have been in a fortunate position to observe it as it evolved and to watch it as it matured and developed. I have a great fondness for the field, and I would like to see it continue to flourish and grow. Even though computer science has emerged as a mature discipline, strong leadership and direction within the profession are still of great importance in order for it to contribute fully to science and society.

What has impressed me most during my years in computer science is the level of commitment of the participants. In the early years, I saw strong individual commitment. Later, I saw institutions joining forces with individuals to form strong support systems for computer science. Today, technology is advancing the discipline so rapidly that the combined commitment of individuals and institutions is no

longer adequate to meet the challenges created by the expansion of knowledge. The demands of industrial research laboratories and academic institutions far surpass the current resources for producing the needed pool of talented researchers. To reap the maximum benefit from the scientific and technological advances of computing, a national commitment must be made and sustained.

Before expanding on this need, let me first relate some of the events in my career that have brought me to this position. When I received my Ph.D. in electrical engineering from Stanford University, my education had included only one course in computing, which was taught by David Huffman. My last year at Stanford coincided with the year Huffman spent there, and it was from him that I received a basic introduction to switching circuits, logic design, and the theory of computation. During the spring of that same year, Edward McCluskey was recruiting faculty for his Digital Systems Laboratory at Princeton. By chance, he happened to be telephoning Bernard Widrow at Stanford to discuss prospective Ph.D. candidates just as I was walking by Widrow's door. When Widrow saw me, he motioned me into his office and handed me the telephone, and we arranged that I would visit Princeton. The fact that I had no formal education in computer science, except for Huffman's course, did not deter McCluskey. At that time, few people had an educational background in computing. After my visit, I was sufficiently impressed by McCluskey's commitment to computer science and with the opportunity he pre-

sented me to begin a career in the developing science of computing that I accepted his job offer as soon as it was extended. This decision significantly altered my career plans, for prior to McCluskey's telephone call, I had planned to teach electrical engineering on the West Coast.

My arrival at Princeton in the fall of 1964 occurred at a time when a dramatic change was taking place in the computing field. Much of the course content in computer science had focused on the design of circuits for digital computers and minimizing the number of transistors needed to build these circuits. By the mid sixties, however, technology had advanced to the point where transistors were about to be replaced by computer chips with as many as a hundred components per chip. Thus, minimizing the number of transistors was no longer relevant. As you can imagine, this had profound ramifications for what was then called computer science; existing courses were about to become obsolete, and new ones had to be developed.

Princeton asked me to develop a course in automata theory to expand the scope of the curriculum beyond the digital circuit design course then being offered. Since there were no courses or books on the subject, I asked McCluskey to recommend some materials for a course on automata theory. He was not sure himself, but he gave me a list of six papers and told me that the material would probably give students a good background in automata theory. McCluskey's list included works by Warren McCulloch and Walter Pitts, John Backus and Peter Naur, Noam Chomsky, Michael Rabin and Dana Scott, Juris Hartmanis and Richard Stearns, and, of course, Alan Turing.

At the time, I thought it strange that individuals were prepared to introduce courses into the curriculum without clearly understanding their content. In retrospect, I realize that people who believe in the future of a subject and who sense its importance will invest in the subject long before they can delineate its boundaries.

When I look back on the material in that early course in automata theory, I am struck by the diversity of these sources. In 1943 McCulloch and Pitts, working in neurophysiology, published a paper on a logical calculus for describing events in neuron nets. These events were series of electrical pulses and could be viewed as strings of zeros and ones. The paper had a notation for describing how these strings of zeros and ones combine in neurons to produce new strings of zeros and ones. This notation was subsequently developed into the language of regular expressions for describing sets of strings.

Rabin and Scott were mathematicians who developed a model of a computer with a finite amount of memory. They called this model the finite-state automaton, and showed that the possible behaviors of finite-state automata were precisely those behaviors that could be described by the regular expressions that grew out of the work of McCulloch and Pitts. This confluence of ideas from two widely different disciplines helped convince early computer scientists of the importance of regular expressions and finite automata.

Chomsky, a linguist, had been studying the syntax of natural languages. In the course of his work, he developed the concept of a context-free grammar. At about the same time, two computer scientists, Backus and Naur, were attempting to develop formalisms for describing programming languages. Before 1960 programming languages were defined by lengthy and often incomplete verbal descriptions. Inconsistencies in various implementations of a language often made it difficult to change software between systems. Backus and Naur developed a formal notation for describing the syntax of various programming languages. Amazingly enough, their notation was equivalent to the context-free grammars developed by Chomsky.

Turing had in 1936 introduced a simple model of a computing device, which is now known as the Turing machine. This device was simple enough that there could be no question that any function computed by it was computable. However, Turing argued further that his model could compute every function considered computable. Simply put, any computational process that could be carried out could be programmed on the Turing machine. Today this hypothesis is universally accepted, and the Turing machine is the foundation of modern computability theory.

Turing's work might have remained in the realm of mathematics and logic were it not for a seminal paper on the computational complexity of algorithms by mathematicians Hartmanis and Stearns. They measured the complexity of an algorithm by the number of steps needed for its execution and used this method to develop a theory of complexity classes. This paper sparked the imagination of many computer scientists and led to the establishment of complexity theory as an integral part of the discipline.

Certain research papers are important not only for their technical contributions, but, more importantly, because they provide a conceptual view or establish a paradigm for research. The work of Hartmanis and Stearns attracted researchers and focused attention

on the topic of complexity. Among the more significant advances that resulted were the classification of the complexity of most major mathematical theories, the reducibility of many combinatorial problems, the concept of NP-completeness, and a deeper understanding of concepts such as randomness.

And so the early automata theory course served to emphasize two important aspects of computer science: the way it has used a multiplicity of ideas from diverse fields to develop and expand, and the way basic research can compound and escalate advances in computing. On a more personal note, the course had a tremendous impact on my career. Through it I met Jeffrey Ullman and Alfred Aho, with whom I subsequently collaborated for many years. *Formal Languages and Automata Theory*, which I wrote with Ullman, also evolved from this course.

In the spring of 1967, Princeton asked me to run a seminar series. As is often the case, funds for the seminars were limited. The intention was to invite local speakers; however, there was just enough money to sponsor two outside speakers from within a 200-mile radius. Drawing from my interest in automata theory, I invited Chomsky and Hartmanis, both of whom agreed to speak. Hartmanis's visit had a significant impact on my career. During the course of his visit, he asked me about prospective Ph.D. candidates for faculty positions at Cornell. Our ensuing discussion that evening on the importance of the science base for computing and the fact that Cornell had established an independent computer science department led me to ask if I might be considered for one of the positions, instead of a new Ph.D. candidate. After visiting Cornell, I immediately accepted their job offer.

I had first learned of Cornell's commitment to computer science from a news article that had appeared the year before Hartmanis's visit—1966. At that time computer science departments were rapidly being established at a rate that exceeded the supply of faculty to staff them. The article discussed Cornell's recognition of this problem and their efforts to solve it. Three faculty members from the mathematics and engineering departments had persuaded the Sloan Foundation to donate one million dollars to develop an independent computer science department to produce the Ph.D.'s needed to staff the computer science departments being created at other institutions.

By the time I arrived at Cornell in 1967, automata theory and complexity theory were established as parts of the computer science curriculum. And so shortly after switching institutions, I decided to switch fields and began to work on algorithms and

data structures. I believed that the methodology of theoretical computer science could be used to develop a science base for algorithmic design that would be useful to the practitioner.

During the 1960s, research on algorithms had been very unsatisfying. A researcher would publish an algorithm in a journal along with execution times for a small set of sample problems, and then several years later, a second researcher would give an improved algorithm along with execution times for the same set of sample problems. The new algorithm would invariably be faster, since in the intervening years, both computer performance and programming languages had improved. The fact that the algorithms were run on different computers and programmed in different languages made me uncomfortable with the comparison. It was difficult to factor out both the effects of increased computer performance and the programming skills of the implementors—to discover the effects due to the new algorithm as opposed to its implementation. Furthermore, it was possible that the second researcher had inadvertently tuned his or her algorithm to the sample problems. Conceivably, if the two algorithms were run again on another sample set of problems, the original algorithm might outperform the newer one.

I set out to demonstrate that a theory of algorithm design based on worst-case asymptotic performance could be a valuable aid to the practitioner. First, a size was associated with each instance of a problem; then the complexity of an algorithm was measured by calculating the rate of growth in computing time as a function of the problem size. Owing to problems in estimating input distribution, the worst-case analysis over all inputs of a given size was adopted as the measure of complexity.

There were a number of problems associated with worst-case asymptotic complexity. Some of the asymptotically optimal algorithms were numerically unstable; others were sufficiently complex that they were unlikely to be programmed. Further, the expected time for realistic input distributions would be a more reasonable measure of complexity. But the paradigm of worst-case asymptotic complexity provided a mathematical criterion for measuring the goodness of an algorithm, making it possible to ask such questions as "What is the optimal algorithm for a problem?" and "What is the intrinsic complexity of a problem?" The idea met with much resistance. People argued that faster computers would remove the need for asymptotic efficiency. Just the opposite is true, however, since as computers become faster, the size of the attempted problems becomes larger,

thereby making asymptotic efficiency even more important.

In early 1970 I took a year-long sabbatical at Stanford University, where I met and shared an office with Robert Tarjan, a second-year graduate student. The research recognized by the 1986 Turing Award took place during that period of collaboration. We worked together on a number of graph algorithms, including graph connectivity and planarity testing. We adopted the philosophy of designing for worst-case performance and demonstrated that a few simple techniques could be used to construct efficient algorithms in many areas of computer applications. We were able to prove that some of our algorithms had worst-case performance optimal within a constant factor, and also that the performance of the resulting algorithms far surpassed that of existing algorithms. Our planarity algorithm was able to test graphs with 1000 vertices in about 10 seconds—two orders of magnitude faster than existing algorithms. Our results attracted many researchers whose efforts created new data structures and design techniques. Today this area is an integral part of computer science. Again, it was not one specific algorithm that was of fundamental importance, but rather that our work captured the imagination of others. It also attracted the attention of bright young researchers who went on to establish data structures and algorithms as an important subdiscipline.

I would like to make some observations and recommendations for the future of computer science. During my career, I have seen the commitment of individuals and institutions grow and expand as computer science has grown and expanded. I believe, however, that much of the credit for the emergence of computer science as a discipline rests with the dedication and commitment of a relatively small number of researchers who had a vision of the potential of computing and the perseverance to make this vision a reality. It is now our responsibility to formulate a new vision, to shape the goals for the next generation of researchers. It is important that computer scientists share this new vision and make it a reality.

Today, computers have penetrated almost every aspect of modern life. They are used in agriculture, communication, education, manufacturing, and medicine. They are used to predict weather, to optimize food production, to control satellites in space, to develop new drugs, and to manufacture fuel-efficient automobiles. In medicine, their use in tomography allows precise imaging of vital organs to aid diagnosis and treatment. In the business world, they are used to route messages, handle commercial

transactions, and access databases. They are advancing the frontiers of knowledge in physics, chemistry, and biology. Computers will also play a vital role in the economic revitalization of this country.

Computers have already made a major impact on the way we think and live. However, I envision an even greater impact. The potential of computer science, if fully explored and developed, will take us to a higher plane of knowledge about the world. Computer science will assist us in gaining a greater understanding of intellectual processes. It will enhance our knowledge of the learning process, the thinking process, and the reasoning process. Computer science will provide models and conceptual tools for the cognitive sciences. Just as the physical sciences have dominated man's intellectual endeavors during this century as researchers explored the nature of matter and the beginning of the universe, today we are beginning the exploration of the intellectual universe of ideas, knowledge structures, and language. I foresee significant advances continuing to be made that will greatly alter our lives. The work Tarjan and I started in the 1970s has led to an understanding of data structures and how to manipulate them. Looking ahead, I can foresee an understanding of how to organize and manipulate knowledge.

In the sixties, a change in technology broadened the scope of computer science from circuit design to computation. The tremendous increases in computing power that are on the horizon today will once again fundamentally change computer science. No one can foresee the precise details, but we can sense the potential for understanding the semantics of language, abstraction, and knowledge representation. Just as early researchers were willing to invest their energies in computer science before fully understanding the details, we must also commit ourselves to the future of computer science before fully discerning its shape.

Today, there are signs that computer science is turning to applications areas. As it contributes its models, tools, and techniques to these new fields, they in turn will contribute new ideas and methodologies that will greatly enrich and expand the scope of computer science. Potentially, we are at the threshold of a new era of growth of the science. However, two areas impede our progress over this threshold.

First there is the inadequate size of the science base, as well as the lack of sufficient researchers to expand it. Despite a sizable body of knowledge, tools, and techniques, the science base of computer science is not developing as rapidly as it should. With computer technology advancing so quickly, it

is easy to lose sight of the importance of developing the underlying science base for computing. There is too great a temptation to focus simply on writing software. As we build larger and more complex systems, we must develop the conceptual tools that will allow us to comprehend the essence of a task and to develop the software tools needed to create complete systems. We have not been able to utilize fully the benefits of computing precisely because we have failed to develop the science base necessary for constructing reliable and user-friendly software systems quickly and economically. Existing programming languages are not satisfactory for the task; they appear to be requiring higher and higher skill levels. We must find ways of communicating with computers that lower the required skill level, just as the assembly line lowered the skill level needed in the production process. A factor of 100 improvement in software productivity would allow us to design, implement, and install a given system in two or three days rather than over the course of a year. Such improvements depend on the development of a science base for programming environments, software development, and man-machine interfaces.

Clearly, the science base established by early researchers has been beneficial. Each of the six papers covered in that early automata theory course added considerably to it. For example, extensions of the work of Backus and Naur on formal descriptions of programming languages allow us to automate the construction of the parsing component of compilation. An early Fortran compiler took 20–50 man-years of effort. Today we assign undergraduate term projects that involve constructing a compiler for a language that is conceptually far more sophisticated than Fortran and see it completed within the term. This dramatic improvement is attributable to the development of a science base in formal languages. This same science base allows us to construct compiler compilers and to tailor languages to specific needs. It allows a chemist to write in a language where the data items are molecules and valences rather than integers and strings. Similar contributions have been made in databases, concurrency, algorithms, VLSI design tools, and many other areas. But greater efforts must be made.

Why is the science base lagging behind the technological base? The field of computer science has grown explosively, more rapidly than any other discipline in history. It is unique in that it evolved from researchers from diverse backgrounds instead of emerging from an existing discipline. Other fields, such as molecular biology, had the advantage of emerging from broader disciplines that could contribute researchers of all ages, along with resources

and structures. Computer scientists came from many backgrounds and have not been able to bring the support structures of a mother discipline with them. Consequently, computer science began without a sufficient number of recognized senior scientists and without existing support structures to facilitate and channel its growth.

The demand for computer science researchers is ever increasing. Even within the profession, there is competition for researchers. The demands of educational institutions and research laboratories for computer scientists are growing faster than the field can build the infrastructure necessary for producing the needed pool of talent. The shortfall in computer science faculty, in particular, has serious ramifications. Unless an investment is made to provide sufficient researchers in our educational institutions, we will fall even further behind in trying to meet this ever-growing demand for computer scientists.

Universities that realized the importance of computer science early on and acted were able to establish departments of the highest quality. Generally, it has been difficult for institutions that started later to catch up. A similar situation is arising on the national scene. Today, there is a global struggle for technological and economic leadership. Computing will play a key role in this struggle. Unless we develop a national policy to support computer science, we will, by our own inaction, allow other countries to establish leads in computing that cannot be overcome.

A national commitment to computer science must be made and sustained. We must develop innovative programs to provide an adequate talent pool of researchers, use existing researchers more effectively, and develop environments to foster research in all aspects of computing. We must increase public awareness of computer science. And we must convince policymakers of the importance of a full-scale commitment to computer science.

CR Categories and Subject Descriptors: A.0 [General Literature]: General—biographies/autobiographies; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems; G.2.2 [Discrete Mathematics]: Graph Theory; K.2 [Computing Milieux]: History of Computing—people

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: John E. Hopcroft, Robert E. Tarjan, Turing Award

Author's Present Address: John E. Hopcroft, Dept. of Computer Science, Cornell University, Ithaca, NY 14853.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Reflections on Software Research

Can the circumstances that existed in Bell Labs that nurtured the UNIX project be produced again?

DENNIS M. RITCHIE

The UNIX¹ operating system has suddenly become news, but it is not new. It began in 1969 when Ken Thompson discovered a little-used PDP-7 computer and set out to fashion a computing environment that he liked. His work soon attracted me; I joined in the enterprise, though most of the ideas, and most of the work for that matter, were his. Before long, others from our group in the research area of AT&T Bell Laboratories were using the system; Joe Ossanna, Doug McIlroy, and Bob Morris were especially enthusiastic critics and contributors. In 1971, we acquired a PDP-11, and by the end of that year we were supporting our first real users: three typists entering patent applications. In 1973, the system was rewritten in the C language, and in that year, too, it was first described publicly at the Operating Systems Principles conference; the resulting paper [8] appeared in *Communications of the ACM* the next year.

Thereafter, its use grew steadily, both inside and outside of Bell Laboratories. A development group was established to support projects inside the company, and several research versions were licensed for outside use.

The last research distribution was the seventh edition system, which appeared in 1979; more recently, AT&T began to market System III, and now offers System V, both products of the development group. All research versions were "as is," unsupported software;

System V is a supported product on several different hardware lines, most recently including the 3B systems designed and built by AT&T.

UNIX is in wide use, and is now even spoken of as a possible industry standard. How did it come to succeed?

There are, of course, its technical merits. Because the system and its history have been discussed at some length in the literature [6, 7, 11], I will not talk about these qualities except for one: despite its frequent surface inconsistency, so colorfully annotated by Don Norman in his *Datamation* article [4] and despite its richness, UNIX is a simple, coherent system that pushes a few good ideas and models to the limit. It is this aspect of the system, above all, that endears it to its adherents.

Beyond technical considerations, there were sociological forces that contributed to its success. First, it appeared at a time when alternatives to large, centrally administered computation centers were becoming possible; the 1970s were the decade of the minicomputer. Small groups could set up their own computation facilities. Because they were starting afresh, and because manufacturers' software was, at best, unimaginative and often horrible, some adventuresome people were willing to take a chance on a new and intriguing, even though unsupported, operating system.

Second, UNIX was first available on the PDP-11, one of the most successful of the new minicomputers that appeared in the 1970s, and soon its portability brought

¹UNIX is a trademark of AT&T Bell Laboratories.

© 1984 ACM 0001-0782/84/0800-0758 \$5.00

it to many new machines as they appeared. At the time that UNIX was created, we were pushing hard for a machine, either a DEC PDP-10 or SDS (later Xerox) Sigma 7. It is certain, in retrospect, that if we had succeeded in acquiring such a machine, UNIX might have been written but would have withered away. Similarly, UNIX owes much to Multics [5], as I have described [6, 7] it eclipsed its parent as much because it does not demand unusual hardware support as because of any other qualities.

Finally, UNIX enjoyed an unusually long gestation period. During much of this time (say 1969–1979), the system was effectively under the control of its designers and being used by them. It took time to develop all of the ideas and software, but even though the system was still being developed people were using it, both inside Bell Labs, and outside under license. Thus, we managed to keep the central ideas in hand, while accumulating a base of enthusiastic, technically competent users who contributed ideas and programs in a calm, communicative, and noncompetitive environment. Some outside contributions were substantial, for example those from the University of California at Berkeley. Our users were widely, though thinly, distributed within the company, at universities, and at some commercial and government organizations. The system became important in the intellectual, if not yet commercial, marketplace because of this network of early users.

What does industrial computer science research consist of? Some people have the impression that the original UNIX work was a bootleg project, a "skunk works." This is not so. Research workers are supposed to discover or invent new things, and although in the early days we subsisted on meager hardware, we always had management encouragement. At the same time, it was certainly nothing like a development project. Our intent was to create a pleasant computing environment for ourselves, and our hope was that others liked it. The Computing Science Research Center at Bell Laboratories to which Thompson and I belong studies three broad areas: theory; numerical analysis; and systems, languages, and software. Although work for its own sake resulting, for example, in a paper in a learned journal, is not only tolerated but welcomed, there is strong though wonderfully subtle pressure to think about problems somehow relevant to our corporation. This has been so since I joined Bell Labs around 15 years ago, and it should not be surprising; the old Bell System may have seemed a sheltered monopoly, but research has always had to pay its way. Indeed, researchers love to find problems to work on; one of the advantages of doing research in a large company is the enormous range of the puzzles that turn up. For example, theorists may contribute to compiler design, or to LSI algorithms; numerical analysts study charge and current distribution in semiconductors; and, of course, software types like to design systems and write programs that people use. Thus, computer research at Bell Labs has always had a considerable commitment to the world, and does not fear edicts commanding us to be practical.

For some of us, in fact, a principal frustration has been the inability to convince others that our research products can indeed be useful. Someone may invent a new application, write an illustrative program, and put it to use in our own lab. Many such demonstrations require further development and continuing support in order for the company to make best use of them. In the past, this use would have been exclusively inside the Bell System; more recently, there is the possibility of developing a product for direct sale.

For example, some years ago Mike Lesk developed an automated directory-assistance system [3]. The program had an online Bell Labs phone book, and was connected to a voice synthesizer on a telephone line with a tone decoder. One dialed the system, and keyed in a name and location code on the telephone's key pad; it spoke back the person's telephone number and office address (It didn't attempt to pronounce the name). In spite of the hashing through twelve buttons (which, for example, squashed "A," "B," and "C" together), it was acceptably accurate: it had to give up on around 5 percent of the tries. The program was a local hit and well-used. Unfortunately, we couldn't find anyone to take it over, even as a supported service within the company, let alone a public offering, and it was an excessive drain on our resources, so it was finally scrapped. (I chose this example not only because it is old enough not to exacerbate any current squabbles, but also because it is timely: The organization that publishes the company telephone directory recently asked us whether the system could be revived.)

Of course not every idea is worth developing or supporting. In any event, the world is changing: Our ideas and advice are being sought much more avidly than before. This increase in influence has been going on for several years, partly because of the success of UNIX, but, more recently, because of the dramatic alteration of the structure of our company.

AT&T divested its telephone operating companies at the beginning of 1984. There has been considerable public speculation about what this will mean for fundamental research at Bell Laboratories; one report in *Science* [2] is typical. One fear sometimes expressed is that basic research, in general, may languish because it yields insufficient short-term gains to the new, smaller AT&T. The public position of the company is reassuring; moreover, research management at Bell Labs seems to believe deeply, and argues persuasively, that the commitment to support of basic research is deep and will continue [1].

Fundamental research at Bell Labs in physics and chemistry and mathematics may, indeed, not be threatened; nevertheless, the danger it might face, and the case against which it must be prepared to argue, is that of irrelevance to the goals of the company. Computer science research is different from these more traditional disciplines. Philosophically it differs from the physical sciences because it seeks not to discover, explain, or exploit the natural world, but instead to study the properties of machines of human creation. In this it is analogous to mathematics, and indeed the "science" part of computer science is, for the most part, mathe-

matical in spirit. But an inevitable aspect of computer science is the creation of computer programs: objects that, though intangible, are subject to commercial exchange.

More than anything else, the greatest danger to good computer science research today may be excessive relevance. Evidence for the worldwide fascination with computers is everywhere, from the articles on the financial, and even the front pages of the newspapers, to the difficulties that even the most prestigious universities experience in finding and keeping faculty in computer science. The best professors, instead of teaching bright students, join start-up companies, and often discover that their brightest students have preceded them. Computer science is in the limelight, especially those aspects, such as systems, languages, and machine architecture, that may have immediate commercial applications. The attention is flattering, but it can work to the detriment of good research.

As the intensity of research in a particular area increases, so does the impulse to keep its results secret. This is true even in the university (Watson's account [12] of the discovery of the structure of DNA provides a well-known example), although in academia there is a strong counterpressure: Unless one publishes, one never becomes known at all. In industry, a natural impulse of the establishment is to guard proprietary information. Researchers understand reasonable restrictions on what and when they publish, but many will become irritated and flee elsewhere, or start working in less delicate areas, if prevented from communicating their discoveries and inventions in suitable fashion. Research management at Bell Labs has traditionally been sensitive to maintaining a careful balance between company interests and the industrial equivalent of academic freedom. The entrance of AT&T into the computer industry will test, and perhaps strain, this balance.

Another danger is that commercial pressures of one sort or another will divert the attention of the best thinkers from real innovation to exploitation of the current fad, from prospecting to mining a known lode. These pressures manifest themselves not only in the disappearance of faculty into industry, but also in the conservatism that overtakes those with well-paying investments—intellectual or financial—in a given idea. Perhaps this effect explains why so few interesting software systems have come from the large computer companies; they are locked into the existing world. Even IBM, which supports a well-regarded and productive research establishment, has in recent years produced little to cause even a minor revolution in the way people think about computers. The working examples of important new systems seem to have come either from entrepreneurial efforts (Visicalc is a good example) or from large companies, like Bell Labs and most especially Xerox, that were much involved with computers and could afford research into them, but did not regard them as their primary business.

On the other hand, in smaller companies, even the most vigorous research support is highly dependent on market conditions. *The New York Times*, in an article

describing Alan Kay's passage from Atari to Apple, notes the problem: "Mr. Kay . . . said that Atari's laboratories had lost some of the atmosphere of innovation that once attracted some of the finest talent in the industry. "When I left last month it was clear that they would be putting their efforts in the short term," he said . . . "I guess the tree of research must from time to time be refreshed with the blood of bean counters." [9]

Partly because they are new and still immature, and partly because they are a creation of the intellect, the arts and sciences of software abridge the chain, usual in physics and engineering, between fundamental discoveries, advanced development, and application. The inventors of ideas about how software should work usually find it necessary to build demonstration systems. For large systems, and for revolutionary ideas, much time is required: It can be said that UNIX was written in the 70s to distill the best systems ideas of the 60s, and became the commonplace of the 80s. The work at Xerox PARC on personal computers, bitmap graphics, and programming environments [10] shows a similar progression, starting, and coming to fruition a few years later. Time, and a commitment to the long-term value of the research, are needed on the part of both the researchers and their management.

Bell Labs has provided this commitment and more: a rare and uniquely stimulating research environment for my colleagues and me. As it enters what company publications call "the new competitive era," its managers and workers will do well to keep in mind how, and under what conditions, the UNIX system succeeded. If we can keep alive enough openness to new ideas, enough freedom of communication, enough patience to allow the novel to prosper, it will remain possible for a future Ken Thompson to find a little-used CRAY/I computer and fashion a system as creative, and as influential, as UNIX.

REFERENCES

1. Bell Labs: New order augurs well. *Nature* 305, 5933 (Sept. 29, 1983).
2. Bell Labs on the brink. *Science* 221 (Sept. 23, 1983).
3. Lesk, M. E. User-activated BTL directory assistance. Bell Laboratories internal memorandum (1972).
4. Norman, D. A. The truth about UNIX. *Datamation* 27, 12 (1981).
5. Organick, E. I. *The Multics System*. MIT Press, Cambridge, MA, 1972.
6. Ritchie, D. M. UNIX time-sharing system: A retrospective. *Bell Syst Tech. J.* 57, 6 (1978). 1947–1969.
7. Ritchie, D. M. The evolution of the UNIX time-sharing system. In *Language Design and Programming Methodology*, Jeffrey M. Tobias, ed., Springer-Verlag, New York, (1980).
8. Ritchie, D. M. and Thompson, K. The UNIX time-sharing system. *Commun. ACM* 17, 7 (July 1974). 365–375.
9. Sanger, D. E. Key Atari scientist switches to Apple. *The New York Times* 133, 46, 033 (May 3, 1984).
10. Thacker, C. P. et al. Alto, a personal computer. Xerox PARC Technical Report CSL-79-11.
11. Thompson, K. UNIX time-sharing system: UNIX implementation. *Bell Syst. Tech. J.* 57, 6 (1978). 1931–1946.
12. Watson, J. D. *The Double Helix: A Personal Account of the Discovery of the Structure of DNA*. Atheneum Publishers, New York (1968).

Author's Present Address: Dennis M. Ritchie, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1967 ACM Turing Lecture

Computers Then and Now

MAURICE V. WILKES

Cambridge University, Cambridge, England

ABSTRACT: Reminiscences on the early developments leading to large scale electronic computers show that it took much longer than was expected for the first of the more ambitious and fully engineered computers to be completed and prove themselves in practical operation. Comments on the present computer field assess the needs for future development.

KEY WORDS AND PHRASES: Moore School, ENIAC, Mauchly, Eckert, Wilkes, von Neumann, Turing, optimum coding, ultrasonic delay line, Carr, Hopper, automatic programming, compilers, interpreters, history, prediction

CR CATEGORIES: 1.2, 1.3

I do not imagine that many of the Turing lecturers who will follow me will be people who were acquainted with Alan Turing. The work on computable numbers, for which he is famous, was published in 1936 before digital computers existed. Later he became one of the first of a distinguished succession of able mathematicians who have made contributions to the computer field. He was a colorful figure in the early days of digital computer development in England, and I would find it difficult to speak of that period without making some references to him.

Pioneering Days

An event of first importance in my life occurred in 1946, when I received a telegram inviting me to attend in the late summer of that year a course on computers at the Moore School of Electrical Engineering in Philadelphia. I was able to attend the latter part of the course, and a wonderful experience it was. No such course had ever been held before, and the achievements of the Moore School, and other computer pioneers, were known to few. There were 28 students from 20 organizations. The principal instructors were John Mauchly and Presper Eckert. They were fresh from their triumph as designers of the ENIAC, which was the first electronic digital computer, although it did not work on the stored program principle. The scale of this machine would be impressive even today—it ran to over 18,000 vacuum tubes. Although the ENIAC was very successful—and very fast—for the computation of ballistic tables, which was the application for which the project was negotiated, it had severe limitations which greatly restricted its application as a general purpose computing device. In the first place, the program was set up by means of plugs and sockets and switches, and it took a long time to change from one problem to another. In the second place, it had internal storage capacity for 20 numbers only. Eckert and Mauchly appreciated that the main problem was one of storage, and they pro-

Presented at the ACM 20th Anniversary Conference, Washington, D.C., August 1967.

posed for future machines the use of ultrasonic delay lines. Instructions and numbers would be mixed in the same memory in the way to which we are now accustomed. Once the new principles were enunciated, it was seen that computers of greater power than the ENIAC could be built with one tenth the amount of equipment.

Von Neumann was, at that time, associated with the Moore School group in a consultative capacity, although I did not personally become acquainted with him until somewhat later. The computing field owes a very great debt to von Neumann. He appreciated at once the possibilities of what became known as logical design, and the potentialities implicit in the stored program principle. That von Neumann should bring his great prestige and influence to bear was important, since the new ideas were too revolutionary for some, and powerful voices were being raised to say that the ultrasonic memory would not be reliable enough, and that to mix instructions and numbers in the same memory was going against nature.

Subsequent developments have provided a decisive vindication of the principles taught by Eckert and Mauchly in 1946 to those of us who were fortunate enough to be in the course. There was, however, a difficult period in the early 1950s. The first operating stored-program computers were, naturally enough, laboratory models; they were not fully engineered and they by no means exploited the full capability of the technology of the time. It took much longer than people had expected for the first of the more ambitious and fully engineered computers to be completed and prove themselves in practical operation. In retrospect, the period seems a short one; at the time, it was a period of much heart searching and even recrimination.

I have often felt during the past year that we are going through a very similar phase in relation to time sharing. This is a development carrying with it many far-reaching implications concerning the relationship of computers to individual users and to communities, and one that has stirred many people's imaginations. It is now several years since the pioneering systems were demonstrated. Once again, it is taking longer than people expected to pass from experimental systems to highly developed ones that fully exploit the technology that we have available. The result is a period of uncertainty and questioning that closely resembles the earlier period to which I referred. When it is all over, it will not take us long to forget the trials and tribulations that we are now going through.

In ultrasonic memories, it was customary to store up to 32 words end to end in the same delay line. The pulse rate was fairly high, but people were much worried about the time spent in waiting for the right word to come around. Most delay line computers were, therefore, designed so that, with the exercise of cunning, the programmer could place his instructions and numbers in the memory in such a way that the waiting time was minimized. Turing himself was a pioneer in this type of logical design. Similar methods were later applied to computers which used a magnetic drum as their memory and, altogether, the subject of *optimum coding*, as it was called, was a flourishing one. I felt that this kind of human ingenuity was misplaced as a long-term investment, since sooner or later we would have truly random-access memories. We therefore did not have anything to do with optimum coding in Cambridge.

Although a mathematician, Turing took quite an interest in the engineering side of computer design. There was some discussion in 1947 as to whether a cheaper substance than mercury could not be found for use as an ultrasonic delay medium.

Turing's contribution to this discussion was to advocate the use of gin, which he said contained alcohol and water in just the right proportions to give a zero temperature coefficient of propagation velocity at room temperature.

A source of strength in the early days was that groups in various parts of the world were prepared to construct experimental computers without necessarily intending them to be the prototype for serial production. As a result, there became available a body of knowledge about what would work and what would not work, about what it was profitable to do and what it was not profitable to do. While, looking around at the computers commercially available today, one cannot feel that all the lessons were learned, there is no doubt that this diversity of research in the early days has paid good dividends. It is, I think, important that we should have similar diversity today when we are learning how to construct large, multiple-access, multi-programmed, multi-processor computer systems. Instead of putting together components and vacuum tubes to make a computer, we have now to learn how to put together memory modules, processors, and peripheral devices to make a system. I hope that money will be available to finance the construction of large systems intended for research only.

Much of the early engineering development of digital computers was done in universities. A few years ago, the view was commonly expressed that universities had played their part in computer design, and that the matter could now safely be left to industry. I do not think that it is necessary that work on computer design should go on in all universities, but I am glad that some have remained active in the field. Apart from the obvious functions of universities in spreading knowledge, and keeping in the public domain material that might otherwise be hidden, universities can make a special contribution by reason of their freedom from commercial considerations, including freedom from the need to follow the fashion.

Good Language and Bad

Gradually, controversies about the design of computers themselves died down and we all began to argue about the merits or demerits of sophisticated programming techniques; the battle for automatic programming or, as we should now say, for the use of higher level programming languages, had begun. I well remember taking part at one of the early ACM meetings—it must have been about 1953—in a debate on this subject. John Carr was also a speaker and he distinguished two groups of programmers; the first comprised the “primitives,” who believed that all instructions should be written in octal, hexadecimal, or some similar form, and who had no time for what they called fancy schemes, while the second comprised the “space cadets,” who saw themselves as the pioneers of a new age. I hastened to enroll myself as a space cadet, although I remember issuing a warning against relying on interpretive systems, for which there was then something of a vogue, rather than on compilers. (I do not think that the term compiler was then in general use, although it had in fact been introduced by Grace Hopper.)

The serious arguments advanced against automatic programming had to do with efficiency. Not only was the running time of a compiled program longer than that of a hand-coded program, but, what was then more serious, it needed more memory. In other words, one needed a bigger computer to do the same work. We all know

that these arguments, although valid, have not proved decisive, and that people have found that it has paid them to make use of automatic programming. In fact, the spectacular expansion of the computing field during the last few years would otherwise have been impossible. We have now a very similar debate raging about time sharing, and the arguments being raised against it are very similar to those raised earlier against automatic programming. Here again, I am on the side of the space cadets, and I expect the debate to have a similar outcome.

Incidentally, I fear that in that automatic programming debate Turing would have been definitely on the side of the primitives. The programming system that he devised for the pioneering computer at Manchester University was bizarre in the extreme. He had a very nimble brain himself and saw no need to make concessions to those less well-endowed. I remember that he had decided—presumably because someone had shown him a train of pulses on an oscilloscope—that the proper way to write binary numbers was backwards, with the least significant digit on the left. He would, on occasion, carry this over into decimal notation. I well remember that once, during a lecture, when he was multiplying some decimal numbers together on the blackboard to illustrate a point about checking a program, we were all unable to follow his working until we realized that he had written the numbers backwards. I do not think that he was being funny, or trying to score off us; it was simply that he could not appreciate that a trivial matter of that kind could affect anybody's understanding one way or the other.

I believe that in twenty years people will look back on the period in which we are now living as one in which the principles underlying the design of programming languages were just beginning to be understood. I am sorry when I hear well-meaning people suggest that the time has come to standardize on one or two languages. We need temporary standards, it is true, to guide us on our way, but we must not expect to reach stability for some time yet.

The Higher Syntax

A notable achievement of the last few years has been to secure a much improved understanding of syntax and of syntax analysis. This has led to practical advances in compiler construction. An early achievement in this field, not adequately recognized at the time, was the Compiler-Compiler of Brooker and Morris.

People have now begun to realize that not all problems are linguistic in character, and that it is high time that we paid more attention to the way in which data are stored in the computer, that is, to data structures. In his Turing lecture given last year, Alan Perlis drew attention to this subject. At the present time, choosing a programming language is equivalent to choosing a data structure, and if that data structure does not fit the data you want to manipulate then it is too bad. It would, in a sense, be more logical first to choose a data structure appropriate to the problem and then look around for, or construct with a kit of tools provided, a language suitable for manipulating that data structure. People sometimes talk about high-level and low-level programming languages without defining very clearly what they mean. If a high-level programming language is one in which the data structure is fixed and unalterable, and a low-level language is one in which there is some latitude in the

choice of data structures, then I think that we may see a swing toward low-level programming languages for some purposes.

I would, however, make this comment. In a high-level language, much of the syntax, and a large part of the compiler, are concerned with the mechanism of making declarations, the forming of compound statements out of simple statements, and with the machinery of conditional statements. All this is entirely independent of what the statements that really operate on the data do or what the data structure is like. We have, in fact, two languages, one inside the other; an outer language that is concerned with the flow of control, and an inner language which operates on the data. There might be a case for having a standard outer language—or a small number to choose from—and a number of inner languages which could be, as it were, plugged in. If necessary, in order to meet special circumstances, a new inner language could be constructed; when plugged in, it would benefit from the power provided by the outer language in the matter of organizing the flow of control. When I think of the number of special languages that we are beginning to require—for example, for real time control, computer graphics, the writing of operating systems, etc.,—the more it seems to me that we should adopt a system which would save us designing and learning to use a new outer language each time.

The fundamental importance of data structures may be illustrated by considering the problem of designing a single language that would be the preferred language either for a purely arithmetic job or for a job in symbol manipulation. Attempts to produce such a language have been disappointing. The difficulty is that the data structures required for efficient implementation in the two cases are entirely different. Perhaps we should recognize this difficulty as a fundamental one, and abandon the quest for an omnibus language which will be all things to all men.

There is one development in the software area which is, perhaps, not receiving the notice that it deserves. This is the increasing mobility of language systems from one computer to another. It has long been possible to secure this mobility by writing the system entirely in some high-level programming language in wide use such as ALGOL or FORTRAN. This method, however, forces the use of the data structures implicit in the host language and this imposes an obvious ceiling on efficiency.

In order that a system may be readily transferred from one computer to another, other than *via* a host language, the system must be written in the first place in machine-independent form. This would not be the place to go into the various techniques that are available for transferring a suitably constructed system. They include such devices as bootstrapping, and the use of primitives and macros. Frequently the operation of transfer involves doing some work on a computer on which the system is already running. Harry Huskey did much early pioneer work in this subject with the NELIAC system.

There is reason to hope that the new-found mobility will extend itself to operating systems, or at least to substantial parts of them. Altogether, I feel that we are entering a new period in which the inconveniences of basic machine-code incompatibility will be less felt. The increasing use of internal filing systems in which information can be held within the system in alphanumeric, and hence in essentially machine-independent, form, will accentuate the trend. Information so held can be transformed by algorithm to any other form in which it may be required. We must get

used to regarding the machine-independent form as the basic one. We will then be quite happy to attach to our computer systems groups of devices that would now be regarded as fundamentally incompatible; in particular, I believe that in the large systems of the future the processors will not necessarily be all out of the same stable.

Design and Assembly

A feature of the last few years has been an intensive interest in computer graphics. I believe that we in the computer field have long been aware of the utility in appropriate circumstances of graphical means of communication with a computer, but I think that many of us were surprised by the appeal that the subject had to mechanical engineers. Engineers are used to communicating with each other by diagrams and sketches and, as soon as they saw diagrams being drawn on the face of a cathode-ray tube, many of them jumped to the conclusion that the whole problem of using a computer in engineering design had been solved. We, of course, know that this is far from being the case, and that much hard work will be necessary before the potential utility of displays can be realized. The initial reaction of engineers showed us, however, two things that we should not forget. One is that, in the judgment of design engineers, the ordinary means of communicating with a computer are entirely inadequate. The second is that graphical communication in some form or other is of vital importance in engineering as that subject is now conducted; we must either provide the capability in our computer systems, or take on the impossible task of training up a future race of engineers conditioned to think in a different way.

There are signs that the recent growth of interest in computer graphics is about to be followed by a corresponding growth of interest in the manipulation of objects by computers. Several projects in this area have been initiated. The driving force behind them is largely an interest in artificial intelligence. Both the tasks chosen and the programming strategy employed reflect this interest.

My own interest in the subject, however, is more practical. I believe that computer controlled mechanical handling devices have a great future in factories and elsewhere. The production of engineering components has been automated to a remarkable extent, and the coming of numerically-controlled machine tools has enabled quite elaborate components to be produced automatically in relatively small batches. By contrast, much less progress has been made in automating the assembly of components to form complete articles.

The artificial intelligence approach may not be altogether the right one to make to the problem of designing automatic assembly devices. Animals and machines are constructed from entirely different materials and on quite different principles. When engineers have tried to draw inspiration from a study of the way animals work they have usually been misled; the history of early attempts to construct flying machines with flapping wings illustrates this very clearly. My own view is that we shall see, before very long, computer-controlled assembly belts with rows of automatic handling machines arranged alongside them, and controlled by the same computer system. I believe that these handling machines will resemble machine tools rather than fingers and thumbs, although they will be lighter in construction and will rely heavily on feedback from sensing elements of various kinds.

The Next Breakthrough

I suppose that we are all asking ourselves whether the computer as we now know it is here to stay, or whether there will be radical innovations. In considering this question, it is well to be clear exactly what we have achieved. Acceptance of the idea that a processor does one thing at a time—at any rate as the programmer sees it—made programming conceptually very simple, and paved the way for the layer upon layer of sophistication that we have seen develop. Having watched people try to program early computers in which multiplications and other operations went on in parallel, I believe that the importance of this principle can hardly be exaggerated. From the hardware point of view, the same principle led to the development of systems in which a high factor of hardware utilization could be maintained over a very wide range of problems, in other words to the development of computers that are truly general purpose. The ENIAC, by contrast, contained a great deal of hardware, some of it for computing and some of it for programming, and yet, on the average problem, only a fraction of this hardware was in use at any given time.

Revolutionary advances, if they come, must come by the exploitation of the high degree of parallelism that the use of integrated circuits will make possible. The problem is to secure a satisfactorily high factor of hardware utilization, since, without this, parallelism will not give us greater power. Highly parallel systems tend to be efficient only on the problems that the designer had in his mind; on other problems, the hardware utilization factor tends to fall to such an extent that conventional computers are, in the long run, more efficient. I think that it is inevitable that in highly parallel systems we shall have to accept a greater degree of specialization towards particular problem areas than we are used to now. The absolute cost of integrated circuits is, of course, an important consideration, but it should be noted that a marked fall in cost would also benefit processors of conventional design.

One area in which I feel that we must pin our hopes on a high degree of parallelism is that of pattern recognition in two dimensions. Present-day computers are woefully inefficient in this area. I am not thinking only of such tasks as the recognition of written characters. Many problems in symbol manipulation have a large element of pattern recognition in them, a good example being syntax analysis. I would not exclude the possibility that there may be some big conceptual breakthrough in pattern recognition which will revolutionize the whole subject of computing.

Summary

I have ranged over the computer field from its early days to where we are now. I did not start quite at the beginning, since the first pioneers worked with mechanical and electro-mechanical devices, rather than with electronic devices. We owe them, however, a great debt, and their work can, I think, be studied with profit even now.

Surveying the shifts of interest among computer scientists and the ever-expanding family of those who depend on computers in their work, one cannot help being struck by the power of the computer to bind together, in a genuine community of interest, people whose motivations differ widely. It is to this that we owe the vitality and vigor of our Association. If ever a change of name is thought necessary, I hope that the words "computing machinery" or some universally recognized synonym will remain. For what keeps us together is not some abstraction, such as Turing machine, or information, but the actual hardware that we work with every day.

The 1978 ACM Turing Award was presented to Robert W. Floyd by Walter Carlson, Chairman of the Awards Committee, at the ACM Annual Conference in Washington, D. C., December 4.

In making the selection, the General Technical Achievement Award Subcommittee (formerly the Turing Award Subcommittee) cited Professor Floyd for "helping to found the following important subfields of computer science: the theory of parsing, the semantics of programming languages, automatic program verification, automatic program synthesis, and analysis of algorithms."

Professor Floyd, who received both his A.B. and B.S. from the University of Chicago in 1953 and 1958, respectively, is a self-taught computer scientist. His study of computing began in 1956, when as a night-operator for an IBM 650, he found the time to learn about programming between loads of card hoppers.

Floyd implemented one of the first Algol 60 compilers, finishing his work on this project in 1962. In the process, he did some early work on compiler optimization. Subsequently, in the

years before 1965, Floyd systematized the parsing of programming languages. For that he originated the precedence method, the bounded context method, and the production language method of parsing.

In 1966 Professor Floyd presented a mathematical method to prove the correctness of programs. He has offered, over the years, a number of fast useful algorithms. These include (1) the tree-sort algorithm for in-place sorting, (2) algorithms for finding the shortest paths through networks, and (3) algorithms for finding medians and convex hulls. In addition, Floyd has determined the limiting speed of digital addition and the limiting speeds for permuting information in a computer memory. His contributions to mechanical theorem-proving and automatic spelling checkers have also been numerous.

In recent years Professor Floyd has been working on the design and implementation of a programming language primarily for student use. It will be suitable for teaching structured programming systematically to novices and will be nearly universal in its capabilities.

The Paradigms of Programming

Robert W. Floyd
Stanford University



Paradigm(pæ·radim, -dīm)... [a. F. *paradigme*, ad. L. *paradigma*, a. Gr. παραδείγμα pattern, example, f. παραδεῖν·*vai* to exhibit beside, show side by side...]
1. A pattern, exemplar, example.

1752 J. Gill *Trinity* v. 91

The archetype, paradigm, exemplar, and idea,
according to which all things were made.

From the Oxford English Dictionary.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Author's address: Department of Computer Science, Stanford University, Stanford CA 94305.

© 1979 ACM 0001-0782/79/0800-0455 \$00.75.

Today I want to talk about the paradigms of programming, how they affect our success as designers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formulated by Dijkstra [6], Wirth [27, 29], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of simpler subproblems. In programming the solution of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangularized system. This gradual decomposition is continued until the subproblems that arise are simple enough to cope with directly. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the *i*th variable from the *i*th equation. Yet further decomposition would yield a fully detailed algorithm.

The second phase of the structured programming paradigm entails working upward from the concrete objects and functions of the underlying machine to the more abstract objects and functions used throughout the modules produced by the top-down design. In the linear equation example, if the coefficients of the equations are rational functions of one variable, we might first design

a multiple-precision arithmetic representation and procedures, then, building upon them, a polynomial representation with its own arithmetic procedures, etc. This approach is referred to as the method of *levels of abstraction*, or of *information hiding*.

The structured programming paradigm is by no means universally accepted. Its firmest advocates would acknowledge that it does not by itself suffice to make all hard problems easy. Other high level paradigms of a more specialized type, such as branch-and-bound [17, 20] or divide-and-conquer [1, 11] techniques, continue to be essential. Yet the paradigm of structured programming does serve to extend one's powers of design, allowing the construction of programs that are too complicated to be designed efficiently and reliably without methodological support.

I believe that the current state of the art of computer programming reflects inadequacies in our stock of paradigms, in our knowledge of existing paradigms, in the way we teach programming paradigms, and in the way our programming languages support, or fail to support, the paradigms of their user communities.

The state of the art of computer programming was recently referred to by Robert Balzer [3] in these words: "It is well known that software is in a depressed state. It is unreliable, delivered late, unresponsive to change, inefficient, and expensive. Furthermore, since it is currently labor intensive, the situation will further deteriorate as demand increases and labor costs rise." If this sounds like the famous "software crisis" of a decade or so ago, the fact that we have been in the same state for ten or fifteen years suggests that "software depression" is a more apt term.

Thomas S. Kuhn, in *The Structure of Scientific Revolutions* [16], has described the scientific revolutions of the past several centuries as arising from changes in the dominant paradigms. Some of Kuhn's observations seem appropriate to our field. Of the scientific textbooks which present the current scientific knowledge to students, Kuhn writes:

Those texts have, for example, often seemed to imply that the content of science is uniquely exemplified by the observations, laws and theories described in their pages.

In the same way, most texts on computer programming imply that the content of programming is the knowledge of the algorithms and language definitions described in their pages.

Kuhn writes, also:

The study of paradigms, including many that are far more specialized than those named illustratively above, is what mainly prepares the student for membership in the particular scientific community with which he will later practice. Because he there joins men who learned the bases of their field from the same concrete models, his subsequent practice will seldom evoke overt disagreement over fundamentals...

In computer science, one sees several such communities, each speaking its own language and using its own paradigms. In fact, programming languages typically

encourage use of some paradigms and discourage others. There are well defined schools of Lisp programming, APL programming, Algol programming, and so on. Some regard data flow, and some control flow, as the primary structural information about a program. Recursion and iteration, copying and sharing of data structures, call by name and call by value, all have adherents.

Again from Kuhn:

The older schools gradually disappear. In part their disappearance is caused by their members' conversion to the new paradigm. But there are always some men who cling to one or another of the older views, and they are simply read out of the profession, which thereafter ignores their work.

In computing, there is no mechanism for reading such men out of the profession. I suspect they mainly become managers of software development.

Balzer, in his jeremiad against the state of software construction, went on to prophesy that automatic programming will rescue us. I wish success to automatic programmers, but until they clean the stables, our best hope is to improve our own capabilities. I believe the best chance we have to improve the general practice of programming is to attend to our paradigms.

In the early 1960's, parsing of context-free languages was a problem of pressing importance in both compiler development and natural linguistics. Published algorithms were usually both slow and incorrect. John Cocke, allegedly with very little effort, found a fast and simple algorithm [2], based on a now standard paradigm which is the computational form of dynamic programming [1]. The dynamic programming paradigm solves a problem for given input by first iteratively solving it for all smaller inputs. Cocke's algorithm successively found all parsings of all substrings of the input. In this conceptual frame, the problem became nearly trivial. The resulting algorithm was the first to uniformly run in polynomial time.

At around the same time, after several incorrect top-down parsers had been published, I attacked the problem of designing a correct one by inventing the paradigm of finding a hierarchical organization of processors, akin to a human organization of employers hiring and discharging subordinates, that could solve the problem, and then simulating the behavior of this organization [8]. Simulation of such multiple recursive processes led me to the use of recursive routines as a control structure. I later found that other programmers with difficult combinatorial problems, for example Gelernter with his geometry-theorem proving machine [10], had apparently invented the same control structure.

John Cocke's experience and mine illustrate the likelihood that continued advance in programming will require the continuing invention, elaboration, and communication of new paradigms.

An example of the effective elaboration of a paradigm is the work by Shortliffe and Davis on the MYCIN [24] program, which skillfully diagnoses, and recommends medication for, bacterial infections. MYCIN is a

rule-based system, based on a large set of independent rules, each with a testable condition of applicability and a resulting simple action when the condition is satisfied. Davis' TEIRESIAS [5] program modifies MYCIN, allowing an expert user to improve MYCIN's performance. The TEIRESIAS program elaborates the paradigm by tracing responsibility backward from an undesired result through the rules and conditions that permitted it, until an unsatisfactory rule yielding invalid results from valid hypotheses is reached. By this means it has become technically feasible for a medical expert who is not a programmer to improve MYCIN's diagnostic capabilities. While there is nothing in MYCIN which could not have been coded in a traditional branching tree of decisions using conditional transfers, it is the use of the rule-based paradigm, with its subsequent elaboration for self-modification, that makes the interactive improvement of the program possible.

If the advancement of the general art of programming requires the continuing invention and elaboration of paradigms, advancement of the art of the individual programmer requires that he expand his *repertory* of paradigms. In my own experience of designing difficult algorithms, I find a certain technique most helpful in expanding my own capabilities. After solving a challenging problem, I solve it again from scratch, retracing only the *insight* of the earlier solution. I repeat this until the solution is as clear and direct as I can hope for. Then I look for a general rule for attacking similar problems, that *would* have led me to approach the given problem in the most efficient way the first time. Often, such a rule is of permanent value. By looking for such a general rule, I was led from the previously mentioned parsing algorithm based on recursive coroutines to the general method of writing nondeterministic programs [9], which are then transformed by a macroexpansion into conventional deterministic ones. This paradigm later found uses in the apparently unrelated area of problem solving by computers in artificial intelligence, becoming embodied in the programming languages PLANNER [12, 13], MICRO-PLANNER [25], and QA4 [23].

The acquisition of new paradigms by the individual programmer may be encouraged by reading other people's programs, but this is subject to the limitation that one's associates are likely to have been chosen for their compatibility with the local paradigm set. Evidence for this is the frequency with which our industry advertises, not for programmers, but for Fortran programmers or Cobol programmers. The rules of Fortran can be learned within a few hours; the associated paradigms take much longer, both to learn and to unlearn.

Contact with programming written under alien conventions may help. Visiting MIT on sabbatical this year, I have seen numerous examples of the programming power which Lisp programmers obtain from having a single data structure, which is also used as a uniform syntactic structure for all the functions and operations which appear in programs, with the capability to manip-

ulate programs as data. Although my own previous enthusiasm has been for syntactically rich languages like the Algol family, I now see clearly and concretely the force of Minsky's 1970 Turing lecture [19], in which he argued that Lisp's uniformity of structure and power of self reference gave the programmer capabilities whose content was well worth the sacrifice of visual form. I would like to arrive at some appropriate synthesis of these approaches.

It remains as true now as when I entered the computer field in 1956 that everyone wants to design a new programming language. In the words written on the wall of a Stanford University graduate student office, "I would rather write programs to help me write programs than write programs." In evaluating each year's crop of new programming languages, it is helpful to classify them by the extent to which they permit and encourage the use of effective programming paradigms. When we make our paradigms explicit, we find that there are a vast number of them. Cordell Green [11] finds that the mechanical generation of simple searching and sorting algorithms, such as merge sorting and Quicksort, requires over a hundred rules, most of them probably paradigms familiar to most programmers. Often our programming languages give us no help, or even thwart us, in using even the familiar and low level paradigms. Some examples follow.

Suppose we are simulating the population dynamics of a predator-prey system—wolves and rabbits, perhaps. We have two equations:

$$W' = f(W, R)$$

$$R' = g(W, R)$$

which give the numbers of wolves and rabbits at the end of a time period, as a function of the numbers at the start of the period.

A common beginner's mistake is to write:

```
FOR I := - - - DO
  BEGIN
    W := f(W, R);
    R := g(W, R)
  END
```

where *g* is, erroneously, evaluated using the modified value of *W*. To make the program work, we must write:

```
FOR I := - - - DO
  BEGIN
    REAL TEMP;
    TEMP := f(W, R);
    R := g(W, R);
    W := TEMP
  END
```

The beginner is correct to believe we should not have to do this. One of our most common paradigms, as in the predator-prey simulation, is simultaneous assignment of new values to the components of state vectors. Yet hardly any language has an operator for simultaneous assignment. We must instead go through the mechanical, time-wasting, and error-prone operation of introducing

one or more temporary variables and shunting the new values around through them.

Again, take this simple-looking problem:

Read lines of text, until a completely blank line is found. Eliminate redundant blanks between the words. Print the text, thirty characters to a line, without breaking words between lines.

Because both input and output are naturally expressed using multiple levels of iteration, and because the input iterations do not nest with the output iterations, the problem is surprisingly hard to program in most programming languages [14]. Novices take three or four times as long with it as instructors expect, ending up either with an undisciplined mess or with a homemade control structure using explicit increments and conditional execution to simulate some of the desired iterations.

The problem is naturally formulated by decomposition into three communicating coroutines [4], for input, transformation, and output of a character stream. Yet, except for simulation languages, few of our programming languages have a coroutine control structure adequate to allow programming the problem in a natural way.

When a language makes a paradigm convenient, I will say the language *supports* the paradigm. When a language makes a paradigm feasible, but not convenient, I will say the language *weakly supports* the paradigm. As the two previous examples illustrate, most of our languages only weakly support simultaneous assignment, and do not support coroutines at all, although the mechanisms required are much simpler and more useful than, say, those for recursive call-by-name procedures, implemented in the Algol family of languages seventeen years ago.

Even the paradigm of structured programming is at best weakly supported by many of our programming languages. To write down the simultaneous equation solver as one designs it, one should be able to write:

```
MAIN_PROGRAM:  
BEGIN  
  TRIANGULARIZE;  
  BACK_SUBSTITUTE  
END;  
  
BACK_SUBSTITUTE:  
FOR I := N STEP -1 UNTIL 1 DO  
  SOLVE_FOR_VARIABLE(I);  
SOLVE_FOR_VARIABLE(I):  
  -- --  
  -- --  
  
TRIANGULARIZE:  
  -- --  
  -- --  
  
Procedures for multiple-precision arithmetic  
Procedures for rational-function arithmetic  
Declarations of arrays
```

In most current languages, one could not present the main program, procedures, and data declarations in this order. Some preliminary human text-shuffling, of a sort readily mechanizable, is usually required. Further, any variables used in more than one of the multiple-precision

procedures must be global to every part of the program where multiple-precision arithmetic can be done, thereby allowing accidental modification, contrary to the principle of information hiding. Finally, the detailed breakdown of a problem into a hierarchy of procedures typically results in very inefficient code, even though most of the procedures, being called from only one place, could be efficiently implemented by macroexpansion.

A paradigm at an even higher level of abstraction than the structured programming paradigm is the construction of a hierarchy of languages, where programs in the highest level language operate on the most abstract objects, and are translated into programs on the next lower level language. Examples include the numerous formula-manipulation languages which have been constructed on top of Lisp, Fortran, and other languages. Most of our lower level languages fail to fully support such superstructures. For example, their error diagnostic systems are usually cast in concrete, so that diagnostic messages are intelligible only by reference to the translated program on the lower level.

I believe that the continued advance of programming as a craft requires development and dissemination of languages which support the major paradigms of their user's communities. The design of a language should be preceded by enumeration of those paradigms, including a study of the deficiencies in programming caused by discouragement of unsupported paradigms. I take no satisfaction from the extensions of our languages, such as the variant records and powersets of Pascal [15, 28], so long as the paradigms I have spoken of, and many others, remain unsupported or weakly supported. If there is ever a science of programming language design, it will probably consist largely of matching languages to the design methods they support.

I do not want to imply that support of paradigms is limited to our programming languages proper. The entire environment in which we program, diagnostic systems, file systems, editors, and all, can be analyzed as supporting or failing to support the spectrum of methods for design of programs. There is hope that this is becoming recognized. For example, recent work at IRIA in France and elsewhere has implemented editors which are aware of the structure of the program they edit [7, 18, 26]. Anyone who has tried to do even such a simple task as changing every occurrence of X as an identifier in a program without inadvertently changing all the other X's, will appreciate this.

Now I want to talk about what we *teach* as computer programming. Part of our unfortunate obsession with form over content, which Minsky deplored in his Turing lecture [19], appears in our typical choices of what to teach. If I ask another professor what he teaches in the introductory programming course, whether he answers proudly "Pascal" or diffidently "FORTRAN," I know that he is teaching a grammar, a set of semantic rules, and some finished algorithms, leaving the students to discover, on their own, some process of design. Even the

texts based on the structured programming paradigm, while giving direction at the highest level, what we might call the "story" level of program design, often provide no help at intermediate levels, at what we might call the "paragraph" level.

I believe it is possible to explicitly teach a set of systematic methods for all levels of program design, and that students so trained have a large head start over those conventionally taught entirely by the study of finished programs.

Some examples of what we can teach follow.

When I introduce to students the input capabilities of a programming language, I introduce a standard paradigm for interactive input, in the form of a macro-instruction I call PROMPT_READ_CHECK_ECHO, which reads until the input datum satisfies a test for validity, then echoes it on the output file. This macro is, on one level, itself a paradigm of iteration and input. At the same time, since it reads once more often than it says "Invalid data," it instantiates a more general, previously taught paradigm for the loop executed " n and a half times".

```
PROMPT_READ_CHECK_ECHO: arguments are a string PROMPT, a variable V to be read, and a condition BAD which characterizes bad data;
PRINT_ON_TERMINAL(PROMPT);
READ_FROM_TERMINAL(V);
WHILE BAD(V) DO
  BEGIN
    PRINT_ON_TERMINAL("Invalid data");
    READ_FROM_TERMINAL(V)
  END;
  PRINT_ON_FILE(V)
```

It also, on a higher level, instantiates the responsibilities of the programmer toward the user of the program, including the idea that each component of a program should be protected from input for which that component was not designed.

Howard Shrobe and other members of the Programmer's Apprentice group [22] at MIT have successfully taught their novice students a paradigm of broad utility, which they call generate/filter/accumulate. The students learn to recognize many superficially dissimilar problems as consisting of enumerating the elements of a set, filtering out a subset, and accumulating some function of the elements in the subset. The MACLISP language [18], used by the students, supports the paradigm; the students provide only the generator, the filter, and the accumulator.

The predator-prey simulation I mentioned earlier is also an instance of a general paradigm, the state-machine paradigm. The state-machine paradigm typically involves representing the state of the computation by the values of a set of storage variables. If the state is complex, the transition function requires a design paradigm for handling simultaneous assignment, particularly since most languages only weakly support simultaneous assignment. To illustrate, suppose we want to compute:

$$\frac{\pi}{6} = \arcsin\left(\frac{1}{2}\right) = \boxed{\frac{1}{2 \cdot \boxed{1}}} + \boxed{\frac{1}{2^3 \cdot 2 \cdot \boxed{3}}}$$

$$+ \boxed{\frac{1 \cdot 3}{2^5 \cdot 2 \cdot 4 \cdot \boxed{5}}} + \boxed{\frac{1 \cdot 3 \cdot 5}{2^7 \cdot 2 \cdot 4 \cdot 6 \cdot \boxed{7}}} + \dots$$

where I have circled the parts of each summand that are useful in computing the next one on the right. Without describing the entire design paradigm for such processes, a part of the design of the state transition is systematically to find a way to get from

$$Q = \frac{1 \cdot 3}{2^5 \cdot 2 \cdot 4}, \quad C = 5$$

$$S = \frac{1}{2} + \frac{1}{2^3 \cdot 2 \cdot 3} + \frac{1 \cdot 3}{2^5 \cdot 2 \cdot 4 \cdot 5}$$

to

$$Q' = \frac{1 \cdot 3 \cdot 5}{2^7 \cdot 2 \cdot 4 \cdot 6}, \quad C' = 7$$

$$S' = \frac{1}{2} + \dots + \frac{1 \cdot 3 \cdot 5}{2^7 \cdot 2 \cdot 4 \cdot 6 \cdot 7}.$$

The experienced programmer has internalized this step, and in all but the most complex cases does it unconsciously. For the novice, seeing the paradigm explicitly enables him to attack state-machine problems more complex than he could without aid, and, more important, encourages him to identify other useful paradigms on his own.

Most of the classical algorithms to be found in texts on computer programming can be viewed as instances of broader paradigms. Simpson's rule is an instance of extrapolation to the limit. Gaussian elimination is problem solution by recursive descent, transformed into iterative form. Merge sorting is an instance of the divide-and-conquer paradigm. For every such classic algorithm, one can ask, "How could I have invented this," and recover what should be an equally classic paradigm.

To sum up, my message to the serious programmer is: spend a part of your working day examining and refining your own methods. Even though programmers are always struggling to meet some future or past deadline, methodological abstraction is a wise long term investment.

To the teacher of programming, even more, I say: identify the paradigms *you* use, as fully as you can, then teach them explicitly. They will serve your students when Fortran has replaced Latin and Sanskrit as the archetypal dead language.

To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support *my* extending your language into one that *does* support my programming methods, I don't need your shiny new languages; like an old car or house, the old language has limitations I have learned to

live with. To persuade me of the merit of your language, you must show me how to construct programs in it. I don't want to discourage the design of new languages; I want to encourage the language designer to become a serious student of the details of the design process.

Thank you, members of the ACM, for naming me to the company of the distinguished men who are my predecessors as Turing lecturers. No one reaches this position without help. I owe debts of gratitude to many, but especially to four men: to Ben Mittman, who early in my career helped and encouraged me to pursue the scientific and scholarly side of my interest in computing; to Herb Simon, our profession's Renaissance man, whose conversation is an education; to the late George Forsythe, who provided me with a paradigm for the teaching of computing; and to my colleague Donald Knuth, who sets a distinguished example of intellectual integrity. I have also been fortunate in having many superb graduate students from whom I think I have learned as much as I have taught them.

To all of you, I am grateful and deeply honored.

Received April 1979

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass. 1974.
2. Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
3. Balzer, R. Imprecise program specification. Report ISI/RR-75-36, Inform. Sciences Inst., Dec. 1975.
4. Conway, M.E. Design of a separable transition-diagram compiler. *Comm. ACM* 6, 7 (July 1963), 396-408.
5. Davis, R. Interactive transfer of expertise: acquisition of new inference rules. Proc. Int. Joint Conf. on Artif. Intell., MIT, Cambridge, Mass., August 1977, pp. 321-328.
6. Dijkstra, E.W. Notes on structured programming. In *Structured Programming*, O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, New York, 1972, pp. 1-82.
7. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J.J. A structure oriented program editor: A first step towards computer assisted programming. Res. Rep. 114, IRIA, Paris, April 1975.
8. Floyd, R.W. The syntax of programming languages—a survey. *IEEE EC-13*, 4 (Aug. 1964), 346-353.
9. Floyd, R.W. Nondeterministic algorithms. *JACM* 14, 4 (Oct. 1967), 636-644.
10. Gelernter. Realization of a geometry-theorem proving machine. In *Computers and Thought*, E. Feigenbaum and J. Feldman, Eds., McGraw-Hill, New York, 1963, pp. 134-152.
11. Green, C.C., and Barstow, D. On program synthesis knowledge. *Artif. Intell.* 10, 3 (June 1978), 241-279.
12. Hewitt, C. PLANNER: A language for proving theorems in robots. Proc. Int. Joint Conf. on Artif. Intell., Washington, D.C., 1969.
13. Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER... AI TR-258, MIT, Cambridge, Mass., April 1972.
14. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM* 21, 8 (Aug. 1978), 666-677.
15. Jensen, K., and Wirth, N. *Pascal User Manual and Report*. Springer-Verlag, New York, 1978.
16. Kuhn, T.S. *The Structure of Scientific Revolutions*. Univ. of Chicago Press, Chicago, Ill., 1970.
17. Lawler, E., and Wood, D. Branch and bound methods: A survey. *Operations Res.* 14, 4 (July-Aug. 1966), 699-719.
18. MACLISP Manual. MIT, Cambridge, Mass., July 1978.
19. Minsky, M. Form and content in computer science. *Comm. ACM* 17, 2 (April 1970), 197-215.
20. Nilsson, N.J. *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971.
21. Parnas, D. On the criteria for decomposing systems into modules. *Comm. ACM* 15, 12 (Dec. 1972), 1053-1058.
22. Rich, C., and Shrobe, H. Initial report on a LISP programmer's apprentice. *IEEE J. Software Eng. SE-4*, 6 (Nov. 1978), 456-467.
23. Rulifson, J.F., Derkson, J.A., and Waldinger, R.J. QA4: A procedural calculus for intuitive reasoning. Tech. Note 73, Stanford Res. Inst., Menlo Park, Calif., Nov. 1972.
24. Shortliffe, E.H. *Computer-based Medical Consultations: MYCIN*. American Elsevier, New York, 1976.
25. Sussman, G.J., Winograd, T., and Charniak, C. MICRO-PLANNER reference manual. AI Memo 203A, MIT, Cambridge, Mass., 1972.
26. Teitelman, W., et al. INTERLISP manual. Xerox Palo Alto Res. Ctr., 1974.
27. Wirth, N. Program development by stepwise refinement. *Comm. ACM* 14, (April 1971), 221-227.
28. Wirth, N. The programming language Pascal. *Acta Informatica* 1, 1 (1971), 35-63.
29. Wirth, N. *Systematic Programming, an Introduction*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

NIKLAUS WIRTH

1984 ACM A.M. TURING AWARD RECIPIENT

Niklaus Wirth of the Swiss Federal Institute of Technology (ETH) was presented the 1984 ACM A.M. Turing Award at the Association's Annual Conference in San Francisco in October in recognition of his outstanding work in developing a sequence of innovative computer languages: Euler, ALGOL-W, Modula, and Pascal. Pascal, in particular, has become significant pedagogically and has established a foundation for future research in the areas of computer language, systems, and architecture. The hallmarks of a Wirth language are its simplicity, economy of design, and high-quality engineering, which result in a language whose notation appears to be a natural extension of algorithmic thinking rather than an extraneous formalism.

Wirth's ability in language design is complemented by a masterful writing ability. In the April 1971 issue of *Communications of the ACM*, Wirth published a seminal paper on Structured Programming ("Program Development by Stepwise Refinement") that recommended top-down structuring of programs (i.e., successively refining program stubs until the program is fully elaborated). The resulting elegant



Niklaus Wirth

and powerful method of exposition remains interesting reading today even after the furor over Structured Programming has subsided. Two later papers, "Toward a Discipline of Real-Time Programming" and "What Can We Do About the Unnecessary Diversity of Notation" (published in CACM in August and November 1974, respectively), speak to Wirth's consistent and dedicated search for an adequate language formalism.

The Turing Award, the Association's highest recognition of technical contributions to the computing community, honors Alan M. Turing, the English mathematician who defined the computer prototype Turing machine

and helped break German ciphers during World War II.

Wirth received his Ph.D. from the University of California at Berkeley in 1963 and was Assistant Professor at Stanford University until 1967. He is Professor at the ETH Zurich since 1968; from 1982 until 1984 he was Chairman of the Division of Computer Science (Informatik) at ETH. Wirth's recent work includes the design and development of the personal computer Lilith in conjunction with the Modula-2 language. In his lecture, Wirth presents a short history of his major projects, drawing conclusions and highlighting the principles that have guided his work.

FROM PROGRAMMING LANGUAGE DESIGN TO COMPUTER CONSTRUCTION

From NELIAC (via ALGOL 60) to Euler and ALGOL W, to Pascal and Modula-2, and ultimately Lilith, Wirth's search for an appropriate formalism for systems programming yields intriguing insights and surprising results.

NIKLAUS WIRTH

It is a great pleasure to receive the Turing Award, and both gratifying and encouraging to receive appreciation for work done over so many years. I wish to thank ACM for bestowing upon me this prestigious award. It is particularly fitting that I receive it in San Francisco, where my professional career began.

Soon after I received notice of the award, my feeling of joy was tempered somewhat by the awareness of having to deliver the Turing lecture. For someone who is an engineer rather than an orator or preacher, this obligation causes some noticeable anxiety. Foremost among the questions it poses is the following: What do people expect from such a lecture? Some will wish to gain technical insight about one's work, or expect an assessment of its relevance or impact. Others will wish to hear how the ideas behind it emerged. Still others expect a statement from the expert about future trends, events, and products. And some hope for a frank assessment of the present engulfing us, either glorifying the monumental advance of our technology or lamenting its cancerous side effects and exaggerations.

In a period of indecision, I consulted some previous Turing lectures and saw that a condensed report about the history of one's work would be quite acceptable. In order to be not just entertaining, I shall try to summarize what I believe I have learned from the past. This choice, frankly, suits me quite well, because neither do I pretend to know more about the future than most others, nor do I like to be proven wrong afterwards. Also, the art of preaching about current achievements

and misdeeds is not my primary strength. This does not imply that I observe the present computing scene without concern, particularly its tumultuous hassle with commercialism.

Certainly, when I entered the computing field in 1960, it was neither so much in the commercial limelight nor in academic curricula. During my studies at the Swiss Federal Institute of Technology (ETH), the only mention I heard of computers was in an elective course given by Ambros P. Speiser, who later became the president of IFIP. The computer ERMETH developed by him was hardly accessible to ordinary students, and so my initiation to the computing field was delayed until I took a course in numerical analysis at Laval University in Canada. But alas, the Alvac III E machinery was out of order most of the time, and exercises in programming remained on paper in the form of untested sequences of hexadecimal codes.

My next attempt was somewhat more successful: At Berkeley, I was confronted with Harry Huskey's pet machine, the Bendix G-15 computer. Although the Bendix G-15 provided some feeling of success by producing results, the gist of the programming art appeared to be the clever allocation of instructions on the drum. If you ignored the art, your programs could well run slower by a factor of one hundred. But the educational benefit was clear: You could not afford to ignore the least little detail. There was no way to cover up deficiencies in your design by simply buying more memory. In retrospect, the most attractive feature was that every detail of the machine was visible and could be

understood. Nothing was hidden in complex circuitry, silicon, or a magic operating system.

On the other hand, it was obvious that computers of the future had to be more effectively programmable. I therefore gave up the idea of studying how to design hardware in favor of studying how to use it more elegantly. It was my luck to join a research group that was engaged in the development—or perhaps rather improvement—of a compiler and its use on an IBM 704. The language was called NELIAC, a dialect of ALGOL 58. The benefits of such a “language” were quickly obvious, and the task of automatically translating programs into machine code posed challenging problems. This is precisely what one is looking for when engaged in the pursuit of a Doctorate. The compiler, itself written in NELIAC, was a most intricate mess. The subject seemed to consist of 1 percent science and 99 percent sorcery, and this tilt had to be changed. Evidently, programs should be designed according to the same principles as electronic circuits, that is, clearly subdivided into parts with only a few wires going across the boundaries. Only by understanding one part at a time would there be hope of finally understanding the whole.

This attempt received a vigorous starting impulse from the appearance of the report on ALGOL 60. ALGOL 60 was the first language defined with clarity; its syntax was even specified in a rigorous formalism. The lesson was that a clear specification is a necessary but not sufficient condition for a reliable and effective implementation. Contact with Aadrian van Wijngaarden, one of ALGOL’s codesigners, brought out the central theme more distinctly: Could ALGOL’s principles be condensed and crystallized even further?

Thus began my adventures in programming languages. The first experiment led to a dissertation and the language *Euler*—a trip with the bush knife through the jungle of language features and facilities. The result was academic elegance, but not much of practical utility—almost an antithesis of the later data-typed and structured programming languages. But it did create a basis for the systematic design of compilers that, so was the hope, could be extended without loss of clarity to accommodate further facilities.

Euler caught the attention of the IFIP Working Group that was engaged in planning the future of ALGOL. The language ALGOL 60, designed by and for numerical mathematicians, had a systematic structure and a concise definition that were appreciated by mathematically trained people but lacked compilers and support by industry. To gain acceptance, its range of application had to be widened. The Working Group assumed the task of proposing a successor and soon split into two camps. On one side were the ambitious who wanted to erect another milestone in language design, and, on the other, those who felt that time was pressing and that an adequately extended ALGOL 60 would be a productive endeavor. I belonged to this second party and submitted a proposal that lost the election. Thereafter, the pro-

posal was improved with contributions from Tony Hoare (a member of the same group) and implemented on Stanford University’s first IBM 360. The language later became known as ALGOL W and was used in several universities for teaching purposes.

A small interlude in this sizable implementation effort is worth mentioning. The new IBM 360 offered only assembler code and, of course, FORTRAN. Neither particularly were loved, either by me or my graduate students, as a tool for designing a compiler. Hence, I mustered the courage to define yet another language in which the ALGOL compiler would be described: A compromise between ALGOL and the facilities offered by the assembler, it would be a machine language with ALGOL-like statement structures and declarations. Notably, the language was defined in a couple of weeks; I wrote the cross compiler on the Burroughs B-5000 computer within four months, and a diligent student transported it to the IBM 360 within an equal period of time. This preparative interlude helped speed up the ALGOL effort considerably. Although envisaged as serving our own immediate needs and to be discarded thereafter, it quickly acquired its own momentum. PL360 became an effective tool in many places and inspired similar developments for other machines.

Ironically, the success of PL360 was also an indication of ALGOL W’s failure. ALGOL’s range of application had been widened, but as a tool for systems programming, it still had evident deficiencies. The difficulty of resolving many demands with a single language had emerged, and the goal itself became questionable. PL/1, released around this time, provided further evidence to support this contention. The *Swiss army knife* idea has its merits, but if driven to excess, the knife becomes a millstone. Also, the size of the ALGOL-W compiler grew beyond the limits within which one could rest comfortably with the feeling of having a grasp, a mental understanding, of the whole program. The desire for a more concise yet more appropriate formalism for systems programming had not been fulfilled. Systems programming requires an efficient compiler generating efficient code that operates without a fixed, hidden, and large so-called run-time package. This goal had been missed by both ALGOL-W and PL/1, both because the languages were complex and the target computers inadequate.

In the fall of 1967, I returned to Switzerland. A year later, I was able to establish a team with three assistants to implement the language that later became known as *Pascal*. Freed from the constraints of obtaining a committee consensus, I was able to concentrate on including the features I myself deemed essential and excluding those whose implementation effort I judged to be incommensurate with the ultimate benefit. The constraint of severely limited manpower is sometimes an advantage.

Occasionally, it has been claimed that Pascal was designed as a language for teaching. Although this is correct, its use in teaching was not the only goal. In

fact, I do not believe in using tools and formalisms in teaching that are inadequate for any practical task. By today's standards, Pascal has obvious deficiencies for programming large systems, but 15 years ago it represented a sensible compromise between what was desirable and what was effective. At ETH, we introduced Pascal in programming classes in 1972, in fact against considerable opposition. It turned out to be a success because it allowed the teacher to concentrate more heavily on structures and concepts than features and peculiarities, that is, on principles rather than techniques.

Our first Pascal compiler was implemented for the CDC 6000 computer family. It was written in Pascal itself. No PL6000 was necessary, and I considered this a substantial step forward. Nonetheless, the code generated was definitely inferior to that generated by FORTRAN compilers for corresponding programs. Speed is an essential and easily measurable criterion, and we believed the validity of the high-level language concept would be accepted in industry only if the performance penalty were to vanish or at least diminish. With this in mind, a second effort—essentially a one-man effort—was launched to produce a high-quality compiler. The goal was achieved in 1974 by Urs Ammann, and the compiler was thereafter widely distributed and is being used today in many universities and industries. Yet the price was high; the effort to generate good (i.e., not even optimal) code is proportional to the mismatch between language and machine, and the CDC 6000 had certainly not been designed with high-level languages in mind.

Ironically again, the principal benefit turned up where we had least expected it. After the existence of Pascal became known, several people asked us for assistance in implementing Pascal on various other machines, emphasizing that they intended to use it for teaching and that speed was not of overwhelming importance. Thereupon, we decided to provide a compiler version that would generate code for a machine of our own design. This code later became known as *P-code*. The P-code version was easy to construct because the new compiler was developed as a substantial exercise in structured programming by stepwise refinement and therefore the first few refinement steps could be adopted unchanged. Pascal-P proved enormously successful in spreading the language among many users. Had we possessed the wisdom to foresee the dimensions of this movement, we would have put more effort and care into designing and documenting P-code. As it was, it remained a side effort to honor the requests in one concentrated stride. This shows that even with the best intentions one may choose one's goals wrongly.

But Pascal gained truly widespread recognition only after Ken Bowles in San Diego recognized that the P-system could well be implemented on the novel microcomputers. His efforts to develop a suitable environment with integrated compiler, filer, editor, and debugger caused a breakthrough: Pascal became available to

thousands of new computer users who were not burdened with acquired habits or stifled by the urge to stay compatible with software of the past.

In the meantime, I terminated work on Pascal and decided to investigate the enticing new subject of multiprogramming, where Hoare had laid respectable foundations and Brinch Hansen had led the way with his *Concurrent Pascal*. The attempt to distill concrete rules for a multiprogramming discipline quickly led me to formulate them in terms of a small set of programming facilities. In order to put the rules to a genuine test, I embedded them in a fragmentary language, whose name was coined after my principal aim: modularity in program systems. The module later turned out to be the principal asset of this language; it gave the abstract concept of information hiding a concrete form and incorporated a method as significant in uniprogramming as in multiprogramming. Also, *Modula* contained facilities to express concurrent processes and their synchronization.

By 1976, I had become somewhat weary of programming languages and the frustrating task of constructing good compilers for existing computers that were designed for old-fashioned "by-hand" coding. Fortunately, I was given the opportunity to spend a sabbatical year at the research laboratory of Xerox Corporation in Palo Alto, where the concept of the powerful personal workstation had not only originated but was also put into practice. Instead of sharing a large, monolithic computer with many others and fighting for a share via a wire with a 3KHz bandwidth, I now used my own computer placed under my desk over a 15MHz channel. The influence of a 5000-fold increase in anything is not foreseeable; it is overwhelming. The most elating sensation was that after 16 years of working for computers, the computer now seemed to work for me. For the first time, I did my daily correspondence and report writing with the aid of a computer, instead of planning new languages, compilers, and programs for others to use. The other revelation was that a compiler for the language *Mesa*, whose complexity was far beyond that of Pascal, could be implemented on such a workstation. These new working conditions were so many orders of magnitude above what I had experienced at home that I decided to try to establish such an environment there as well.

I finally decided to dig into hardware design. This decision was reinforced by my old disgust with existing computer architectures that made life miserable for a compiler designer with a bent toward systematic simplicity. The idea of designing and building an entire computer system consisting of hardware, microcode, compiler, operating system, and program utilities quickly took shape in my imagination—a design that would be free from any constraint to be compatible with a PDP-11 or an IBM 360, or FORTRAN. Pascal, UNIX, or whatever other current fad or committee standard there might be.

But a sensation of liberation is not enough to succeed

in a technical project. Hard work, determination, a sensitive feeling of what is essential and what ephemeral, and a portion of luck are indispensable. The first lucky accident was a telephone call from a hardware designer enquiring about the possibility of coming to our university to learn about software techniques and acquire a Ph.D. Why not teach him about software and let him teach us about hardware? It didn't take long before the two of us became a functioning team, and Richard Ohran soon became so excited about the new design that he almost totally forgot both software and Ph.D. That didn't disturb me too much, for I was amply occupied with the design of hardware parts; with specifying the micro- and macrocodes, and by programming the latter's interpreter; with planning the overall software system; and in particular with programming a text editor and a diagram editor, both making use of the new high-resolution bit-mapped display and the small miracle called *Mouse* as a pointing device. This exercise in programming highly interactive utility programs required the study and application of techniques quite foreign to conventional compiler and operating system design.

The total project was so diversified and complex that it seemed irresponsible to start it, particularly in view of the small number of part-time assistants available to us, who averaged around seven. The major threat was that it would take too long to keep the enthusiastic two of us persisting and to let the others, who had not yet experienced the power of the workstation idea, become equally enthusiastic. To keep the project within reasonable dimensions, I stuck to three dogmas: Aim for a *single-processor* computer to be operated by a *single user* and programmed in a *single language*. Notably, these cornerstones were diametrically opposed to the trends of the time, which favored research in multiprocessor configurations, time-sharing multiuser operating systems, and as many languages as you could muster.

Under the constraints of a single language, I faced a difficult choice whose effects would be wide ranging, namely, that of selecting a language. Of existing languages, none seemed attractive. Neither could they satisfy all the requirements, nor were they particularly appealing to the compiler designer who knows the task has to be accomplished in a reasonable time span. In particular, the language had to accommodate all our wishes with regard to structuring facilities, based on 10 years' experience with Pascal, and it had to cater to problems so far only handled by coding with an assembler. To cut a long story short, the choice was to design an offspring of both proven Pascal and experimental Modula, that is, *Modula-2*. The module is the key to bringing under one hat the contradictory requirements of high-level abstraction for security through redundancy checking and low-level facilities that allow access to individual features of a particular computer. It lets the programmer encapsulate the use of low-level facilities in a few small parts of the system, thus protecting him from falling into their traps in unexpected places.

The *Lilith* project proved that it is not only possible but advantageous to design a single-language system. Everything from device drivers to text and graphics editors is written in the same language. There is no distinction between modules belonging to the operating system and those belonging to the user's program. In fact, that distinction almost vanishes and with it the burden of a monolithic, bulky resident block of code, which no one wants but everyone has to accept. Moreover, the *Lilith* project proved the benefits of a well-matched hardware/software design. These benefits can be measured in terms of speed: Comparisons of execution times of Modula programs revealed that *Lilith* is often superior to a VAX 750 whose complexity and cost are a multiple of those of *Lilith*. They can also be measured in terms of space: The code of Modula programs for *Lilith* is shorter than the code for PDP-11, VAX, or 68000 by factors of 2 to 3, and shorter than that of the NS 32000 by a factor of 1.5 to 2. In addition, the code generating parts of compilers for these microprocessors are considerably more intricate than they are in *Lilith* due to their ill-matched instruction sets. This length factor has to be multiplied by the inferior density factor, which casts a dark shadow over the much advertised high-level language suitability of modern microprocessors and reveals these claims to be exaggerated. The prospect that these designs will be reproduced millions of times is rather depressing, for by their mere number they become our standard building blocks. Unfortunately, advances in semiconductor technology have been so rapid that architectural advances are overshadowed and have become seemingly less relevant. Competition forces manufacturers to freeze new designs into silicon long before they have proved their effectiveness. And whereas bulky software can at least be modified and at best be replaced, nowadays complexity has descended into the very chips. And there is little hope that we have a better mastery of complexity when we apply it to hardware rather than software.

On both sides of this fence, complexity has and will maintain a strong fascination for many people. It is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. However, this should not diminish our desire for *elegant* solutions, which convince by their clarity and effectiveness. Simple, elegant solutions are more effective, but they are *harder* to find than complex ones, and they require more time, which we too often believe to be unaffordable.

Before closing, let me try to distill some of the common characteristics of the projects that were mentioned. A very important technique that is seldom used as effectively as in computing is the *bootstrap*. We used it in virtually every project. When developing a tool, be it a programming language, a compiler, or a computer, I designed it in such a way that it was beneficial in the very next step: PL360 was developed to implement ALGOL W; Pascal to implement Pascal; Modula-2 to implement the whole workstation software; and *Lilith*

to provide a suitable environment for all our future work, ranging from programming to circuit documentation and development, from report preparation to font design. Bootstrapping is the most effective way of profiting from one's own efforts as well as suffering from one's mistakes.

This makes it mandatory to *distinguish early between what is essential and what ephemeral*. I have always tried to identify and focus in on what is essential and yields unquestionable benefits. For example, the inclusion of a coherent and consistent scheme of data type declarations in a programming language I consider essential, whereas the details of varieties of for-statements, or whether the compiler distinguishes between upper- and lowercase letters, are ephemeral questions. In computer design, I consider the choice of addressing modes and the provision of complete and consistent sets of (signed and unsigned) arithmetic instructions including proper traps on overflow to be crucial; in contrast, the details of a multichannel prioritized interrupt mechanism are rather peripheral. Even more important is ensuring that the ephemeral never impinge on the systematic, structured design of the central facilities. Rather, the ephemeral must be added fittingly to the existing, well-structured framework.

Rejecting pressures to include all kinds of facilities that "might also be nice to have" is sometimes hard. The danger that one's desire to please will interfere with the goal of consistent design is very real. I have always tried to weigh the gains against the cost. For example, when considering the inclusion of either a language feature or the compiler's special treatment of a reasonably frequent construct, one must weigh the benefits against the added cost of its implementation and its mere presence, which results in a larger system. Language designers often fail in this respect. I gladly admit that certain features of *Ada* that have no counterparts in Modula-2 may be nice to have occasionally, but at the same time, I question whether they are worth the price. The price is considerable: First, although the design of both languages started in 1977, Ada compilers have only now begun to emerge, whereas we have been using Modula since 1979. Second, Ada compilers are rumored to be gigantic programs consisting of several hundred thousand lines of code, whereas our newest Modula compiler measures some five thousand lines only. I confess secretly that this Modula compiler is already at the limits of comprehensible complexity, and I would feel utterly incapable of constructing a good compiler for Ada. But even if the effort of building unnecessarily large systems and the cost of memory to contain their code could be ignored, the real cost is hidden in the unseen efforts of the innumerable programmers trying desperately to understand them and use them effectively.

Another common characteristic of the projects sketched was the *choice of tools*. It is my belief that a tool should be commensurate with the product; it must be as simple as possible, but no simpler. A tool is in fact

counterproductive when a large part of the entire project is taken up by mastering the tool. Within the Euler, ALGOL W, and PL360 projects, much consideration was given to the development of table-driven, bottom-up syntax analysis techniques. Later, I switched back to the simple recursive-descent, top-down method, which is easily comprehensible and unquestionably sufficiently powerful, if the syntax of the language is wisely chosen. In the development of the Lilith hardware, we restricted ourselves to a good oscilloscope; only rarely was a logic state analyzer needed. This was possible due to a relatively systematic, trick-free concept for the processor.

Every single project was primarily a *learning experiment*. One learns best when inventing. Only by actually *doing* a development project can I gain enough familiarity with the intrinsic difficulties and enough confidence that the inherent details can be mastered. I never could separate the design of a language from its implementation, for a rigid definition without the feedback from the construction of its compiler would seem to me presumptuous and unprofessional. Thus, I participated in the construction of compilers, circuitry, and text and graphics editors, and this entailed microprogramming, much high-level programming, circuit design, board layout, and even wire wrapping. This may seem odd, but I simply like hands-on experience much better than team management. I have also learned that researchers accept leadership from a factual, in-touch team member much more readily than from an organization expert, be he a manager in industry or a university professor. I try to keep in mind that teaching by setting a good example is often the most effective method and sometimes the only one available.

Lastly, each of these projects was carried through by the enthusiasm and the desire to succeed in the knowledge that the endeavor was worthwhile. This is perhaps the most essential but also the most elusive and subtle prerequisite. I was lucky to have team members who let themselves be infected with enthusiasm, and here is my chance to thank all of them for their valuable contributions. My sincere thanks go to all who participated, be it in the direct form of working in a team, or in the indirect forms of testing our results and providing feedback, of contributing ideas through criticism or encouragement, or of forming user societies. Without them, neither ALGOL W, nor Pascal, nor Modula-2, nor Lilith would have become what they are. This Turing Award also honors their contributions.

Author's Present Address: Niklaus Wirth, Xerox Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

First ACM Turning Lecture

The Synthesis of Algorithmic Systems

ALAN J. PERLIS

Carnegie Institute of Technology, Pittsburgh, Pennsylvania

Introduction

Both knowledge and wisdom extend man's reach. Knowledge led to computers, wisdom to chopsticks. Unfortunately our association is overinvolved with the former. The latter will have to wait for a more sublime day.

On what does and will the fame of Turing rest? That he proved a theorem showing that for a general computing device—later dubbed a “Turing machine”—there existed functions which it could not compute? I doubt it. More likely it rests on the model he invented and employed: his formal mechanism.

This model has captured the imagination and mobilized the thoughts of a generation of scientists. It has provided a basis for arguments leading to theories. His model has proved so useful that its generated activity has been distributed not only in mathematics, but through several technologies as well. The arguments that have been employed are not always formal and the consequent creations not all abstract. Indeed a most fruitful consequence of the Turing machine has been with the creation, study and computation of functions which are computable, i.e., in computer programming. This is not surprising since computers can compute so much more than we yet know how to specify.

I am sure that all will agree that this model has been enormously valuable. History will forgive me for not devoting any attention in this lecture to the effect which Turing had on the development of the general-purpose digital computer, which has further accelerated our involvement with the theory and practice of computation.

Since the appearance of Turing's model there have, of course, been others which have concerned and benefited us in computing. I think, however, that only one has had an effect as great as Turing's: the formal mechanism called ALGOL. Many will immediately disagree, pointing out that too few of us have understood it or used it. While such has, unhappily, been the case, it is not the point. The impulse given by ALGOL to the development of research in computer science is relevant while the number of adherents is not. ALGOL, too, has mobilized our thoughts and has provided us with a basis for our arguments.

I have long puzzled over why ALGOL has been such a useful model in our field. Perhaps some of the reasons are:

- (a) its international sponsorship;
- (b) the clarity of description in print of its syntax;
- (c) the natural way it combines important programmatic features of assembly and subroutine programming;
- (d) the fact that the language is naturally decomposable so that one may suggest

Presented at the 21st ACM National Conference, August 1966.

and define rather extensive modifications to parts of the language without destroying its impressive harmony of structure and notation. There is an appreciated substance to the phrase "ALGOL-like" which is often used in arguments about programming, languages and computation. ALGOL appears to be a durable model, and even flourishes under surgery—be it explorative, plastic or amputative;

(e) the fact that it is tantalizingly inappropriate for many tasks we wish to program.

Of one thing I am sure: ALGOL does not owe its magic to its process of birth: by committee. Thus, we should not be disappointed when eggs, similarly fertilized, hatch duller models. These latter, while illuminating impressive improvements over ALGOL, bring on only a yawn from our collective imaginations. These may be improvements over ALGOL, but they are not successors as models.

Naturally we should and do put to good use the improvements they offer to rectify the weakness of ALGOL. And we should also ponder why they fail to stimulate our creative energies. Why, we should ask, will computer science research, even computer practice, work, but not leap, forward under their influence? I do not pretend to know the whole answer, but I am sure that an important part of their dullness comes from focusing attention on the wrong weaknesses of ALGOL.

The Synthesis of Language and Data Structures

We know that we design a language to simplify the expression of an unbounded number of algorithms created by an important class of problems. The design should be performed only when the algorithms for this class impose, or are likely to impose, after some cultivation, considerable traffic on computers as well as considerable composition time by programmers using existing languages. The language, then, must reduce the cost of a set of transactions to pay its cost of design, maintenance and improvement.

Successor languages come into being from a variety of causes:

- (a) The correction of an error or omission or superfluity in a given language exposes a natural redesign which yields a superior language.
- (b) The correction of an error or omission or superfluity in a given language requires a redesign to produce a useful language.
- (c) From any two existing languages a third can usually be created which (i) contains the facilities of both in an integrated form, and (ii) requires a grammar and evaluation rules less complicated than the collective grammar and evaluation rules of both.

With the above in mind, where might one commence in synthesizing a successor model which will not only improve the commerce with machines but will focus our attention on important problems within computation itself?

I believe the natural starting point must be the organization and classifying of data. It is, to say the least, difficult to create an algorithm without knowing the nature of its data. When we attempt to represent an algorithm in a programming language, we must know the representation of the algorithm's data in that language before we can hope to do a useful computation.

Since our successor is to be a general programming language, it should possess

Synthesis of Algorithmic Systems

general data structures. Depending on how you look at it, this is neither as hard nor as easy as you might think. How should this possession be arranged? Let us see what has been done in the languages we already have. There the approach has been as follows:

- (a) A few "primitive" data structures, e.g., integers, reals, arrays homogeneous in type, lists, strings and files, are defined into the language.
- (b) On these structures a "sufficient" set of operations, e.g., arithmetic, logical, extractive, assignment and combinational, is provided.
- (c) Any other data structure is considered to be nonprimitive and must be represented in terms of primitive ones. The inherent organization in the nonprimitive structures is explicitly provided for by operations over the primitive data, e.g., the relationship between the real and imaginary parts of a complex number by real arithmetic.
- (d) The "sufficient" set of operations for these nonprimitive data structures is organized as procedures.

This process of extension cannot be faulted. Every programming language must permit its facile use, for ultimately it is always required. However, if this process of extension is too extensively used, algorithms often fail to exhibit a clarity of structure which they really possess. Even worse, they tend to execute more slowly than necessary. The former weakness arises because the language was defined the wrong way for the algorithm, while the latter exists because the language forces overorganization in the data and requires administration during execution that could have been done once prior to execution of the algorithm. In both cases, variables have been bound at the wrong time by the syntax and the evaluation rules.

I think that all of us are aware that our languages have not had enough data types. Certainly, in our successor model we should not attempt to remedy this shortcoming by adding a few more, e.g., a limited number of new types and a general catchall structure.

Our experience with the definition of functions should have told us what to do: not to concentrate on a complete set of defined functions at the level of general use, but to provide within the language the structures and control from which the efficient definition and use of functions within programs would follow.

Consequently, we should focus our attention in our successor model on providing the means for defining data structures. But this is not of itself enough. The "sufficient" set of accompanying operations, the contexts in which they occur and their evaluation rules must also then be given within the program for which the data structures are specified.

A list of some of the capabilities that must be provided for data structures would include:

- (a) structure definition;
- (b) assignment of a structure to an identifier, i.e., giving the identifier information cells;
- (c) rules for naming the parts, given the structure;
- (d) assignment of values to the cells attached to an identifier;
- (e) rules for referencing the identifier's attached cells;
- (f) rules of combination, copy and erasure both of structure and cell contents.

These capabilities are certainly now provided in limited form in most languages, but usually in too fixed a way within their syntax and evaluation rules.

We know that the designers of a language cannot fix how much information will reside in structure and how much in the data carried within a structure. Each program must be permitted its natural choice to achieve a desired balance between time and storage. We know there is no single way to represent arrays or list structures or strings or files or combinations of them. The choice depends on

- (a) the frequency of access;
- (b) the frequency of structure changes in which given data is embedded, e.g., appending to a file new record structures or bordering arrays;
- (c) the cost of unnecessary bulk in computer storage requirements;
- (d) the cost of unnecessary time in accessing data; and
- (e) the importance of an algorithmic representation capable of orderly growth so that clarity of structure always exists.

These choices, goodness knows, are difficult for a programmer to make. They are certainly impossible to make at the design level.

Data structures cannot be created out of thin air. Indeed the method we customarily employ is the use of a background machine with fixed, primitive data structures. These structures are those identified with real computers, though the background machine might be more abstract as far as the defining of data structures is concerned. Once the background machine is chosen, additional structure as required by our definitions must be represented as data, i.e., as a name or pointer to a structure. Not all pointers reference the same kind of structure. Since segments of a program are themselves structures, pointers such as "procedure identifier contents of (x)" establish a class of variables whose values are procedure names.

Constants and Variables

Truly, the flexibility of a language is measured by that which programmers may be permitted to vary, either in composition or in execution. The systematic development of variability in language is a central problem in programming and hence in the design of our successor. Always our experience presents us with special cases from which we establish the definition of new variables. Each new experience focuses our attention on the need for more generality. Time sharing is one of our new experiences that is likely to become a habit. Time sharing focuses our attention on the management of our systems and the management by programmers of their texts before, during and after execution. Interaction with program will become increasingly flexible, and our successor must not make this difficult to achieve. The vision we have of conversational programming takes in much more than rapid turn around time and convenient debugging aids: our most interesting programs are never wrong and never final. As programmers we must isolate that which is new with conversational programming before we can hope to provide an appropriate language model for it. I contend that what is new is the requirement to make variable in our languages what we previously had taken as fixed. I do not refer to new data classes now, but to variables whose values are programs or parts of programs, syntax or parts of syntax, and regimes of control.

Most of our attention is now paid to the development of systems for managing

files which improve the administration of the overall system. Relatively little is focused on improving the management of a computation. Whereas the former can be done outside the languages in which we write our programs, for the latter we must improve our control over variability within the programming language we use to solve our problems.

In the processing of a program text an occurrence of a segment of texts may appear in the text once but be executed more than once. This raises the need to identify both constancy and variability. We generally take that which has the form of being variable and make it constant by a process of initialization; and we often permit this process itself to be subject to replication. This process of initialization is a fundamental one and our successor must have a methodical way of treating it.

Let us consider some instances of initialization and variability in ALGOL:

(a) *Entry to a block.* On entry to a block declarations make initializations, but only about some properties of identifiers. Thus, `integer x` initializes the property of being an integer, but it is not possible to initialize the value of `x` as something that will not change during the scope of the block. The declaration `procedure P (. . .); . . . ;` emphatically initializes the identifier `P`, but it is not possible to change it in the block. `array A [1:n, 1:m]` is assigned an initial structure. It is not possible to initialize the values of its cells, or to vary the structure attached to the identifier `A`.

(b) *for statement.* These expressions, which I will call the step and until elements, cannot be initialized.

(c) *Procedure declaration.* This is an initialization of the procedure identifier. On a procedure call, its formal parameters are initialized as procedure identifiers are, and they may even be initialized as to value. However, different calls establish different initializations of the formal parameter identifiers but not different initialization patterns of the values.

The choice permitted in ALGOL in the binding of form and value to identifiers has been considered adequate. However, if we look at the operations of assignment of form, evaluation of form and initialization as important functions to be rationally specified in a language, we might find ALGOL to be limited and even capricious in its available choices. We should expect the successor to be far less arbitrary and limited.

Let me give a trivial example. In the *for* statement the use of a construct such as `value E`, where `E` is an expression, as a step element would signal the initialization of the expression `E`. `value` is a kind of operator that controls the binding of `value` to a form. There is a natural scope attached to each application of the operator.

I have mentioned that procedure identifiers are initialized through declaration. Then the attachment of procedure to identifier can be changed by assignment. I have already mentioned how this can be done by means of pointers. There are, of course, other ways. The simplest is not to change the identifier at all, but rather to have a selection index that picks a procedure out of a set. The initialization now defines an array of forms, e.g., `procedure array P [1:k] (f1, f2, . . . , fk); . . . begin . . . end; . . . ; begin . . . end;` The call `P [i] (a1, a2, . . . , ai)` would select the *i*th procedure body for execution. Or one could define a `procedure switch P := A, B, C` and procedure designational expressions so that the above call would select the *i*th procedure designational expression for execution. The above approaches are too static for some applications and they lack an important property of

assignment: the ability to determine when an assigned form is no longer accessible so that its storage may be otherwise used. A possible application for such procedures, i.e., ones that are dynamically assigned, is as generators. Suppose we have a procedure for computing (a) $\sum_{k=0}^N C_k(N)X^k$ as an approximation to some function

(b) $f(x) = \sum_{k=0}^{\infty} C_k X^k$, when the integer N is specified. Now once having found the

$C_k(N)$, we are merely interested in evaluating (a) for different values of x . We might then wish to define a procedure which prepares (a) from (b). This procedure, on its initial execution, assigns, either to itself or to some other identifier, the procedure which computes (a). Subsequent calls on that identifier will only yield this created computation. Such dynamic assignment raises a number of attractive possibilities:

(a) Some of the storage for the program can be released as a consequence of the second assignment.

(b) Data storage can be assigned as the own of the procedure identifier whose declaration or definition is created.

(c) The initial call can modify the resultant definition, e.g., call by name or call by value of a formal parameter in the initial call will effect the kind of definition obtained.

It is easy to see that the point I am getting at is the necessity of attaching a uniform approach to initialization and the variation of form and value attached to identifiers. This is a requirement of the computation process. As such our successor language must possess a general way of commanding the actions of initialization and variation for its classes of identifiers.

One of the actions we wish to perform in conversational programming is the systematic, or controlled, modification of values of data and text, as distinguished from the unsystematic modification which occurs in debugging. The performance of such actions clearly implies that certain pieces of a text are understood to be variable. Again we accomplish this by declaration, by initialization and by assignment. Thus we may write, in a block heading, the declarations

```
real x, s;
arithmetic expression t, u;
```

In the accompanying text the occurrence of $s := x + t$; causes the value of the arithmetic expression assigned to t , e.g., by input, to be added to that of x and the result assigned as the value of s . We observe that t may have been entered and stored as a form. The operation $+$ can then only be accomplished after a suitable transfer function shall have been applied. The fact that a partial translation of the expression is all that can be done at the classical "translate time" should not deter us. It is time that we began to face the problems of partial translation in a systematic way. The natural pieces of text which can be variable are those identified by the syntactic units of the language.

It is somewhat more difficult to arrange for unpremeditated variation of programs. Here the major problems are the identification of the text to be varied in the original text, and how to find its correspondent under the translation process in the text actually being evaluated. It is easy to say: execute the original text interpretively. But it is through intermediate solutions lying between translation and interpretation that the satisfactory balance of costs is to be found. I should like to express a point

of view in the next section which may shed some light on achieving this balance as each program requires it.

Data Structure and Syntax

Even though list structures and recursive control will not play a central role in our successor language, it will owe a great deal to LISP. This language induces humorous arguments among programmers, often being damned and praised for the same feature. I should only like to point out here that its description consciously reveals the proper components of language definition with more clarity than any language I know of. The description of LISP includes not only its syntax, but the representation of its syntax as a data structure of the language, and the representation of the environment data structure also as a data structure of the language. Actually the description hedges somewhat on the latter description, but not in any fundamental way. From the foregoing descriptions it becomes possible to give a description of the evaluation process as a LISP program using a few primitive functions. While this completeness of description is possible with other languages, it is not generally thought of as part of their defining description.

An examination of ALGOL shows that its data structures are not appropriate for representing ALGOL texts, at least not in a way appropriate for descriptions of the language's evaluation scheme. The same remark may be made about its inappropriateness for describing the environmental data structure of ALGOL programs.

I regard it as critical that our successor language achieve the balance of possessing the data structures appropriate to representing syntax and environment so that the evaluation process can be clearly stated in the language.

Why is it so important to give such a description? Is it merely to attach to the language the elegant property of "closure" so that bootstrapping can be organized? Hardly. It is the key to the systematic construction of programming systems capable of conversational computing.

A programming language has a syntax and a set of evaluation rules. They are connected through the representation of programs as data to which the evaluation rules apply. This data structure is the internal or evaluation directed syntax of the language. We compose programs in the external syntax which, for the purposes of human communication, we fix. The internal syntax is generally assumed to be so translator and machine dependent that it is almost never described in the literature. Usually there is a translation process which takes text from an external to an internal syntax representation. Actually the variation in the internal description is more fundamentally associated with the evaluation rules than the machine on which it is to be executed. The choice of evaluation rules depends in a critical way on the binding time of the variables of the language.

This points out an approach to the organization of evaluation useful in the case of texts which change. Since the internal data structure reflects the variability of the text being processed, let the translation process choose the appropriate internal representation of the syntax, and a general evaluator select specific evaluation rules on the basis of the syntax structure chosen. Thus we must give clues in the external syntax which indicate the variable. For example, the occurrence of **arithmetic expression t; real u,v;** and the statement $u := v/3*t$; indicates the possibility of a different internal syntax for $v/3$ and the value of t . It should be pointed out that t

behaves very much like an ALGOL formal parameter. However, the control over assignment is less regimented. I think this merely points out that formal-actual assignments are independent of the closed subroutine concept and that they have been united in the procedure construct as a way of specifying the scope of an initialization.

In the case of unpremeditated change a knowledge of the internal syntax structure makes possible the least amount of retranslation and alteration of the evaluation rules when text is varied.

Since one has to examine and construct the data structures and evaluation rules entirely in some language, it seems reasonable that it be in the source language itself. One may define as the target of translation an internal syntax whose character strings are a subset of those permitted in the source language. Such a syntax, if chosen to be close to machine code, can then be evaluated by rules which are very much like those of a machine.

While I have spoken glibly about variability attached to the identifiers of the language, I have said nothing about the variability of control. We do not really have a way of describing control, so we cannot declare its regimes. We should expect our successor to have the kinds of control that ALGOL has—and more. Parallel operation is one kind of control about which much study is being done. Another one just beginning to appear in languages is the distributed control, which I will call monitoring. Process A continuously monitors process B so that when B attains a certain state, A intervenes to control the future activity of the process. The control within A could be written **when** P **then** S ; P is a predicate which is always, within some defining scope, under test. Whenever P is **true**, the computation under surveillance is interrupted and S is executed. We wish to mechanize this construct by testing P whenever an action has been performed which could possibly make P **true**, but not otherwise. We must then, in defining the language, the environment and the evaluation rules, include the states which can be monitored during execution. From these primitive states others can be constructed by programming. With a knowledge of these primitive states, arrangement for splicing in testing at possible points can be done even before the specific predicates are defined within a program. We may then trouble-shoot our programs without disturbing the programs themselves.

Variation of the Syntax

Within the confines of a single language an astonishing amount of variability is attainable. Still all experience tells us that our changing needs will place increasing pressure on the language itself to change. The precise nature of these changes cannot be anticipated by designers, since they are the consequence of programs yet to be written for problems not yet solved. Ironically, it is the most useful and successful languages that are most subject to this pressure for change. Fortunately, the early kind of variation to be expected is somewhat predictable. Thus, in scientific computing the representation and arithmetic of numbers varies, but the nature of expressions does not change except through its operands and operators. The variation in syntax from these sources is quite easily taken care of. In effect the syntax and evaluation rules of arithmetic expression are left undefined in the language. Instead

syntax and evaluation rules are provided in the language for programming the definition of arithmetic expression, and to set the scope of such definitions.

The only real difficulty in this one-night-stand language definition game is the specification of the evaluation rules. They must be given with care. For example, in introducing this way the arithmetic of matrices, the evaluation of matrix expressions should be careful of the use of temporary storage and not perform unnecessary iterations.

A natural technique to employ in the use of definitions is to start with a language X , consider the definitions as enlarging the syntax to that of a language X' and give the evaluation rules as a reduction process which reduces any text in X' to an equivalent one in X .

It should be remarked that the variation of the syntax requires a representation of the syntax, preferably as a data structure of X itself.

Conclusion

Programming languages are built around the variable—its operations, control and data structures. Since these are concepts common to all programming, general language must focus on their orderly development. While we owe a great debt to Turing for his simple model, which also focused on the important concepts, we do not hesitate to operate with more sophisticated machines and data than he found necessary. Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily. Our progress, then, is measured by the balance we achieve between efficiency and generality. As the nature of our involvement with computation changes—and it does—the appropriate description of language changes; our emphasis shifts. I feel that our successor model will show such a change. Computer science is a restless infant and its progress depends as much on shifts in point of view as on the orderly development of our current concepts.

None of the ideas presented here are new; they are just forgotten from time to time.

I wish to thank the Association for the privilege of delivering this first Turing lecture. And what better way is there to end this lecture than to say that if Turing were here today he would say things differently in a lecture named differently.

1973 ACM Turing Award Lecture

The Turing Award citation read by Richard G. Canning, chairman of the 1973 Turing Award Committee, at the presentation of this lecture on August 28 at the ACM Annual Conference in Atlanta:

A significant change in the computer field in the last five to eight years has been made in the way we treat and handle data. In the early days of our field, data was intimately tied to the application programs that used it. Now we see that we want to break that tie. We want data that is independent of the application programs that use it—that is, data that is organized and structured to serve many applications and many users. What we seek is the data base.

This movement toward the data base is in its infancy. Even so, it appears that there are now between 1,000 and 2,000 true data base management systems installed worldwide. In ten years very likely, there will be tens of thousands of such systems. Just from the quantities of installed systems, the impact of data bases promises to be huge.

This year's recipient of the A.M. Turing Award is one of the real pioneers of data base technology. No other individual has had the influence that he has had upon this aspect of our field. I

single out three prime examples of what he has done. He was the creator and principal architect of the first commercially available data base management system—the Integrated Data Store—originally developed from 1961 to 1964.^{1,2,3,4} I-D-S is today one of the three most widely used data base management systems. Also, he was one of the founding members of the CODASYL Data Base Task Group, and served on that task group from 1966 to 1968. The specifications of that task group are being implemented by many suppliers in various parts of the world.^{5,6} Indeed, currently these specifications represent the only proposal of stature for a common architecture for data base management systems. It is to his credit that these specifications, after extended debate and discussion, embody much of the original thinking of the Integrated Data Store. Thirdly, he was the creator of a powerful method for displaying data relationships—a tool for data base designers as well as application system designers.^{7,8}

His contributions have thus represented the union of imagination and practicality. The richness of his work has already had, and will continue to have, a substantial influence upon our field.

I am very pleased to present the 1973 A.M. Turing Award to Charles W. Bachman.

The Programmer as Navigator

by Charles W. Bachman



This year the whole world celebrates the five-hundredth birthday of Nicolaus Copernicus, the famous Polish astronomer and mathematician. In 1543, Copernicus published his book, *Concerning the Revolutions of Celestial Spheres*, which described a new theory about the relative physical movements of the earth, the planets, and the sun. It was in direct contradiction with the earth-centered theories which had been established by Ptolemy 1400 years earlier.

Copernicus proposed the heliocentric theory, that planets revolve in a circular orbit around the sun. This theory was subjected to tremendous and persistent criticism. Nearly 100 years later, Galileo was ordered

to appear before the Inquisition in Rome and forced to state that he had given up his belief in the Copernican theory. Even this did not placate his inquisitors, and he was sentenced to an indefinite prison term, while Copernicus's book was placed upon the Index of Prohibited Books, where it remained for another 200 years.

I raise the example of Copernicus today to illustrate a parallel that I believe exists in the computing or, more properly, the information systems world. We have spent the last 50 years with almost Ptolemaic information systems. These systems, and most of the thinking about systems, were based on a "computer centered" concept. (I choose to speak of 50 years of history rather than 25, for I see today's information systems as dating from the beginning of effective punched card equipment rather than from the beginning of the stored program computer.)

Just as the ancients viewed the earth with the sun revolving around it, so have the ancients of our information systems viewed a tab machine or computer with a sequential file flowing through it. Each was an

Copyright © 1973, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address: Honeywell Information Systems, Inc., 200 Smith Street, Waltham, MA 02154.

The abstract, key words, etc., are on page 654.

¹⁻⁸ Footnotes are on page 658.

adequate model for its time and place. But after a while, each has been found to be incorrect and inadequate and has had to be replaced by another model that more accurately portrayed the real world and its behavior.

Copernicus presented us with a new point of view and laid the foundation for modern celestial mechanics. That view gave us the basis for understanding the formerly mysterious tracks of the sun and the planets through the heavens. A new basis for understanding is available in the area of information systems. It is achieved by a shift from a computer-centered to the database-centered point of view. This new understanding will lead to new solutions to our database problems and speed our conquest of the *n*-dimensional data structures which best model the complexities of the real world.

The earliest databases, initially implemented on punched cards with sequential file technology, were not significantly altered when they were moved, first from punched card to magnetic tape and then again to magnetic disk. About the only things that changed were the size of the files and the speed of processing them.

In sequential file technology, search techniques are well established. Start with the value of the primary data key, of the record of interest, and pass each record in the file through core memory until the desired record, or one with a higher key, is found. (A primary data key is a field within a record which makes that record unique within the file.) Social security numbers, purchase order numbers, insurance policy numbers, bank account numbers are all primary data keys. Almost without exception, they are synthetic attributes specifically designed and created for the purpose of uniqueness. Natural attributes, e.g. names of people and places, dates, time, and quantities, are not assuredly unique and thus cannot be used.

The availability of direct access storage devices laid the foundation for the Copernican-like change in viewpoint. The directions of "in" and "out" were reversed. Where the input notion of the sequential file world meant "into the computer from tape," the new input notion became "into the database." This revolution in thinking is changing the programmer from a stationary viewer of objects passing before him in core into a mobile navigator who is able to probe and traverse a database at will.

Direct access storage devices also opened up new ways of record retrieval by primary data key. The first was called randomizing, calculated addressing, or hashing. It involved processing the primary data key with a specialized algorithm, the output of which identified a preferred storage location for that record. If the record sought was not found in the preferred location, then an overflow algorithm was used to search places where the record alternately would have been stored, if it existed at all. Overflow is created when the preferred location is full at the time the record was originally stored.

As an alternative to the randomizing technique, the

Copernicus completely reoriented our view of astronomical phenomena when he suggested that the earth revolves about the sun. There is a growing feeling that data processing people would benefit if they were to accept a radically new point of view, one that would liberate the application programmer's thinking from the centralism of core storage and allow him the freedom to act as a navigator within a database. To do this, he must first learn the various navigational skills; then he must learn the "rules of the road" to avoid conflict with other programmers as they jointly navigate the database information space.

This reorientation will cause as much anguish among programmers as the heliocentric theory did among ancient astronomers and theologians.

Key Words and Phrases: access method, attributes, calculated addressing, celestial mechanics, clustering, contamination, database, database key, database set, deadlock, deadly embrace, entity, hash addressing, overflow, owner, member, primary data key, Ptolemy, relationship, retrieval, secondary data key, sequential file, set, shared access, update, Weyerhaeuser

CR Categories: 3.74, 4.33, 4.34, 5.6, 8.1

index sequential access technique was developed. It also used the primary data key to control the storage and retrieval of records, and did so through the use of multilevel indices.

The programmer who has advanced from sequential file processing to either index sequential or randomized access processing has greatly reduced his access time because he can now probe for a record without sequentially passing all the intervening records in the file. However, he is still in a one-dimensional world as he is dealing with only one primary data key, which is his sole means of controlling access.

From this point, I want to begin the programmer's training as a full-fledged navigator in an *n*-dimensional data space. However, before I can successfully describe this process, I want to review what "database management" is.

It involves all aspects of storing, retrieving, modifying, and deleting data in the files on personnel and production, airline reservations, or laboratory experiments —data which is used repeatedly and updated as new information becomes available. These files are mapped through some storage structure onto magnetic tapes or disk packs and the drives that support them.

Database management has two main functions. First is the inquiry or retrieval activity that reaccesses previously stored data in order to determine the recorded status of some real world entity or relationship. This data has previously been stored by some other job, seconds, minutes, hours, or even days earlier, and has been held in trust by the database management system. A database management system has a continuing re-

sponsibility to maintain data between the time when it was stored and the time it is subsequently required for retrieval. This retrieval activity is designed to produce the information necessary for decision making.

Part of the inquiry activity is report preparation. In the early years of sequential access storage devices and the resultant batch processing, there was no viable alternative to the production of massive file dumps as formatted as reports. Spontaneous requirements to examine a particular checking account balance, an inventory balance, or a production plan could not be handled efficiently because the entire file had to be passed to extract any data. This form of inquiry is now diminishing in relative importance and will eventually disappear except for archival purposes or to satisfy the appetite of a parkinsonian bureaucracy.

The second activity of database management is to update, which includes the original storage of data, its repeated modification as things change, and ultimately, its deletion from the system when the data is no longer needed.

The updating activity is a response to the changes in the real world which must be recorded. The hiring of a new employee would cause a new record to be stored. Reducing available stock would cause an inventory record to be modified. Cancelling an airline reservation would cause a record to be deleted. All of these are recorded and updated in anticipation of future inquiries.

The sorting of files has been a big user of computer time. It was used in sorting transactions prior to batch sequential update and in the preparation of reports. The change to transaction-mode updating and on-demand inquiry and report preparation is diminishing the importance of sorting at the file level.

Let us now return to our story concerning the programmer as navigator. We left him using the randomizing or the index sequential technique to expedite either inquiry or update of a file based upon a primary data key.

In addition to a record's primary key, it is frequently desirable to be able to retrieve records on the basis of the value of some other fields. For example, it may be desirable, in planning ten-year awards, to select all the employee records with the "year-of-hire" field value equal to 1964. Such access is retrieval by secondary data key. The actual number of records to be retrieved by a secondary key is unpredictable and may vary from zero to possibly include the entire file. By contrast, a primary data key will retrieve a maximum of one record.

With the advent of retrieval on secondary data keys, the previously one-dimensional data space received additional dimensions equal to the number of fields in the record. With small or medium-sized files, it is feasible for a database system to index each record in the file on every field in the record. Such totally indexed files are classified as inverted files. In large active files, however, it is not economical to index every field. Therefore, it is prudent to select the fields whose con-

tent will be frequently used as a retrieval criterion and to create secondary indices for those fields only.

The distinction between a file and a database is not clearly established. However, one difference is pertinent to our discussion at this time. In a database, it is common to have several or many different kinds of records. For an example, in a personnel database there might be employee records, department records, skill records, deduction records, work history records, and education records. Each type of record has its own unique primary data key, and all of its other fields are potential secondary data keys.

In such a database the primary and secondary keys take on an interesting relationship when the primary key of one type of record is the secondary key of another type of record. Returning to our personnel database as an example—the field named "department code" appears in both the employee record and the department record. It is one of several possible secondary data keys of the employee records and the single primary data key of the department records.

This equality of primary and secondary data key fields reflects real world relationships and provides a way to reestablish these relationships for computer processing purposes. The use of the same data value as a primary key for one record and as a secondary key for a set of records is the basic concept upon which data structure sets are declared and maintained. The Integrated Data Store (I-D-S) systems and all other systems based on its concepts consider their basic contribution to the programmer to be the capability to associate records into data structure sets and the capability to use these sets as retrieval paths. All the COBOL Database Task Group systems implementations fall into this class.

There are many benefits gained in the conversion from several files, each with a single type of record, to a database with several types of records and database sets. One such benefit results from the significant improvement in performance that accrues from using the database sets in lieu of both primary and secondary indices to gain access to all the records with a particular data key value. With database sets, all redundant data can be eliminated, reducing the storage space required. If redundant data is deliberately maintained to enhance retrieval performance at the cost of maintenance, then the redundant data can be controlled to ensure that the updating of a value in one record will be properly reflected in all other appropriate records. Performance is enhanced by the so-called "clustering" ability of databases where the owner and some or most of the members records of a set are physically stored and accessed together on the same block or page. These systems have been running in virtual memory since 1962.

Another significant functional and performance advantage is to be able to specify the order of retrieval of the records within a set based upon a declared sort field or the time of insertion.

In order to focus the role of programmer as navigator, let us enumerate his opportunities for record access. These represent the commands that he can give to the database system—singly, multiply or in combination with each other—as he picks his way through the data to resolve an inquiry or to complete an update.

1. He can start at the beginning of the database, or at any known record, and sequentially access the "next" record in the database until he reaches a record of interest or reaches the end.
2. He can enter the database with a database key that provides direct access to the physical location of a record. (A database key is the permanent virtual memory address assigned to a record at the time that it was created.)
3. He can enter the database in accordance with the value of a primary data key. (Either the indexed sequential or randomized access techniques will yield the same result.)
4. He can enter the database with a secondary data key value and sequentially access all records having that particular data value for the field.
5. He can start from the owner of a set and sequentially access all the member records. (This is equivalent to converting a primary data key into a secondary data key.)
6. He can start with any member record of a set and access either the next or prior member of that set.
7. He can start from any member of a set and access the owner of the set, thus converting a secondary data key into a primary data key.

Each of these access methods is interesting in itself, and all are very useful. However, it is the synergistic usage of the entire collection which gives the programmer great and expanded powers to come and go within a large database while accessing only those records of interest in responding to inquiries and updating the database in anticipation of future inquiries.

Imagine the following scenario to illustrate how processing a single transaction could involve a path through the database. The transaction carries with it the primary data key value or database key of the record that is to be used to gain an entry point into the database. That record would be used to gain access to other records (either owner or members) of a set. Each of these records is used in turn as a point of departure to examine another set.

For example, consider a request to list the employees of a particular department when given its departmental code. This request could be supported by a database containing only two different types of records: personnel records and department records. For simplicity purposes, the department record can be envisioned as having only two fields: the department code, which is the primary data key; and the department name, which is descriptive. The personnel record can be envisioned as having only three fields: the employee number, which

is the primary data key for the record; the employee name, which is descriptive; and the employee's department code, which is a secondary key which controls set selection and the record's placement in a set. The joint usage of the department code by both records and the declaration of a set based upon this data key provide the basis for the creation and maintenance of the set relationship between a department record and all the records representing the employees of that department. Thus the usage of the set of employee records provides the mechanism to readily list all the employees of a particular department following the primary data key retrieval of the appropriate department record. No other record for index need be accessed.

The addition of the department manager's employee number to the department record greatly extends the navigational opportunities, and provides the basis for a second class of sets. Each occurrence of this new class includes the department records for all the departments managed by a particular employee. A single employee number or department code now provides an entry point into an integrated data structure of an enterprise. Given an employee number, and the set of records of departments managed, all the departments which he manages can be listed. The personnel of each such department can be further listed. The question of departments managed by each of these employees can be asked repeatedly until all the subordinate employees and departments have been displayed. Inversely, the same data structure can easily identify the employee's manager, the manager's manager, and the manager's manager's manager, and so on, until the company president is reached.

There are additional risks and adventures ahead for the programmer who has mastered operation in the n -dimensional data space. As navigator he must brave dimly perceived shoals and reefs in his sea, which are created because he has to navigate in a shared database environment. There is no other obvious way for him to achieve the required performance.

Shared access is a new and complex variation of multiprogramming or time sharing, which were invented to permit shared, but independent, use of the computer resources. In multiprogramming, the programmer of one job doesn't know or care that his job might be sharing the computer, as long as he is sure that his address space is independent of that of any other programs. It is left to the operating system to assure each program's integrity and to make the best use of the memory, processor, and other physical resources. Shared access is a specialized version of multiprogramming where the critical, shared resources are the records of the database. The database records are fundamentally different than either main storage or the processor because their data fields change value through update and do not return to their original condition afterward. Therefore, a job that repeatedly uses a database record may find that record's content or set mem-

bership has changed since the last time it was accessed. As a result, an algorithm attempting a complex calculation may get a somewhat unstable picture. Imagine attempting to converge on an iterative solution while the variables are being randomly changed! Imagine attempting to carry out a trial balance while someone is still posting transactions to the accounts! Imagine two concurrent jobs in an airline reservations system trying to sell the last seat on a flight!

One's first reaction is that this shared access is nonsense and should be forgotten. However, the pressures to use shared access are tremendous. The processors available today and in the foreseeable future are expected to be much faster than are the available direct access storage devices. Furthermore, even if the speed of storage devices were to catch up with that of the processors, two more problems would maintain the pressure for successful shared access. The first is the trend toward the integration of many single purpose files into a few integrated databases; the second is the trend toward interactive processing where the processor can only advance a job as fast as the manually created input messages allow. Without shared access, the entire database would be locked up until a batch program or transaction and its human interaction had terminated.

The performance of today's direct access storage devices is greatly affected by patterns of usage. Performance is quite slow if the usage is an alternating pattern of: access, process, access, process, . . . , where each access depends upon the interpretation of the prior one. When many independent accesses are generated through multiprogramming, they can often be executed in parallel because they are directed toward different storage devices. Furthermore, when there is a queue of requests for access to the same device, the transfer capacity for that device can actually be increased through seek and latency reduction techniques. This potential for enhancing throughput is the ultimate pressure for shared access.

Of the two main functions of database management, inquiry and update, only update creates a potential problem in shared access. An unlimited number of jobs can extract data simultaneously from a database without trouble. However, once a single job begins to update the database, a potential for trouble exists. The processing of a transaction may require the updating of only a few records out of the thousands or possibly millions of records within a database. On that basis, hundreds of jobs could be processing transactions concurrently and actually have no collisions. However, the time will come when two jobs will want to process the same record simultaneously.

The two basic causes of trouble in shared access are interference and contamination. *Interference* is defined as the negative effect of the updating activity of one job upon the results of another. The example I have given of one job running an accounting trial balance while another was posting transactions illustrates the inter-

ference problem. When a job has been interfered with, it must be aborted and restarted to give it another opportunity to develop the correct output. Any output of the prior execution must also be removed because new output will be created. *Contamination* is defined as the negative effect upon a job which results from a combination of two events: when another job has aborted and when its output (i.e. changes to the database or messages sent) has already been read by the first job. The aborted job and its output will be removed from the system. Moreover, the jobs contaminated by the output of the aborted job must also be aborted and restarted so that they can operate with correct input data.

A critical question in designing solutions to the shared access problem is the extent of visibility that the application programmer should have. The Weyerhaeuser Company's shared access version of I-D-S was designed on the premise that the programmer should not be aware of shared access problems. That system automatically blocks each record updated and every message sent by a job until that job terminates normally, thus eliminating the contamination problem entirely. One side effect of this dynamic blocking of records is that a deadlock situation can be created when two or more jobs each want to wait for the other to unblock a desired record. Upon detecting a deadlock situation, the I-D-S database system responds by aborting the job that created the deadlock situation, by restoring the records updated by that job, and by making those records available to the jobs waiting. The aborted job, itself, is subsequently restarted.

Do these deadlock situations really exist? The last I heard, about 10 percent of all jobs started in Weyerhaeuser's transaction-oriented system had to be aborted for deadlock. Approximately 100 jobs per hour were aborted and restarted! Is this terrible? Is this too inefficient? These questions are hard to answer because our standards of efficiency in this area are not clearly defined. Furthermore, the results are application-dependent. The Weyerhaeuser I-D-S system is 90 percent efficient in terms of jobs successfully completed. However, the real questions are:

—Would the avoidance of shared access have permitted more or fewer jobs to be completed each hour?

—Would some other strategy based upon the detecting rather than avoiding contamination have been more efficient?

—Would making the programmer aware of shared access permit him to program around the problem and thus raise the efficiency?

All these questions are beginning to impinge on the programmer as navigator and on the people who design and implement his navigational aids.

My proposition today is that it is time for the application programmer to abandon the memory-centered view, and to accept the challenge and opportunity of navigation within an *n*-dimensional data space. The software systems needed to support such capabilities

exist today and are becoming increasingly available.

Bertrand Russell, the noted English mathematician and philosopher, once stated that the theory of relativity demanded a change in our imaginative picture of the world. Comparable changes are required in our imaginative picture of the information system world.

The major problem is the reorientation of thinking of data processing people. This includes not only the programmer but includes the application system designers who lay out the basic application programming tasks and the product planners and the system programmers who will create tomorrow's operating system, message system, and database system products.

Copernicus laid the foundation for the science of celestial mechanics more than 400 years ago. It is this science which now makes possible the minimum energy solutions we use in navigating our way to the moon and the other planets. A similar science must be developed which will yield corresponding minimum energy solutions to database access. This subject is doubly interesting because it includes the problems of traversing an existing database, the problems of how to build one in the first place and how to restructure it later to best fit the changing access patterns. Can you imagine restructuring our solar system to minimize the travel time between the planets?

It is important that these mechanics of data structures be developed as an engineering discipline based upon sound design principles. It is important that it can be taught and is taught. The equipment costs of the database systems to be installed in the 1980's have been estimated at \$100 billion (at 1970 basis of value). It has further been estimated that the absence of effective

Footnotes to the Turing Award citation on page 653 are:

¹ A general purpose programming system for random access memories (with S.B. Williams). Proc. AFIPS 1964 FJCC, Vol. 26, AFIPS Press, Montvale, N.J., pp. 411-422.

² Integrated Data Store. *DPMA Quarterly* (Jan. 1965).

³ Software for random access processing. *Datamation* (Apr. 1965), 36-41.

⁴ Integrated Data Store—Case Study. Proc. Sec. Symp. on Computer-Centered Data Base Systems sponsored by ARPA, SDC, and ESD, 1966.

⁵ Implementation techniques for data structure sets. Proc. of SHARE Working Conf. on Data Base Systems, Montreal, Canada, July 1973.

⁶ The evolution of data structures. Proc. NordDATA Conf., Aug. 1973, Copenhagen, Denmark, pp. 1075-1093.

⁷ Data structure diagrams. Data Base 1, 2 (1969), Quarterly Newsletter of ACM SIGBDP, pp. 4-10.

⁸ Set concepts for data structures. In *Encyclopedia of Computer Science*, Amerback Corp. (to be published in 1974).

Related articles are:

The evolution of storage structures. *Comm. ACM* 15, 7 (July 1972), 628-634.

Architectural Definition Technique: its objectives, theory, process, facilities and practice (with J. Bouvard). Proc. 1972 ACM SIGFIDET workshop on Data Description, Access and Control, pp. 257-280.

Data space mapped into three dimensions; a viable model for studying data structures. Data Base Management Rep., InfoTech Information Ltd., Berkshire, U.K., 1973.

A direct access system with procedurally generated data structuring capability (with S. Brewer). *Honeywell Comput. J.* (to appear).

standardization could add 20 percent or \$20 billion to the bill. Therefore, it is prudent to dispense with the conservatism, the emotionalism, and the theological arguments which are currently slowing progress. The universities have largely ignored the mechanics of data structures in favor of problems which more nearly fit a graduate student's thesis requirement. Big database systems are expensive projects which university budgets simply cannot afford. Therefore, it will require joint university/industry and university/government projects to provide the funding and staying power necessary to achieve progress. There is enough material for a half dozen doctoral theses buried in the Weyerhaeuser system waiting for someone to come and dig it out. By this I do not mean research on new randomizing algorithms. I mean research on the mechanics of nearly a billion characters of real live business data organized in the purest data structures now known.

The publication policies of the technical literature are also a problem. The ACM SIGBDP and SIGFIDET publications are the best available, and membership in these groups should grow. The refereeing rules and practices of Communications of the ACM result in delays of one year to 18 months between submittal and publication. Add to that the time for the author to prepare his ideas for publication and you have at least a two-year delay between the detection of significant results and their earliest possible publication.

Possibly the greatest single barrier to progress is the lack of general database information within a very large portion of the computer users resulting from the domination of the market by a single supplier. If this group were to bring to bear its experience, requirements, and problem-solving capabilities in a completely open exchange of information, the rate of change would certainly increase. The recent action of SHARE to open its membership to all vendors and all users is a significant step forward. The SHARE-sponsored Working Conference on Database Systems held in Montreal in July (1973) provided a forum so that users of all kinds of equipment and database systems could describe their experiences and their requirements.

The widening dialog has started. I hope and trust that we can continue. If approached in this spirit, where no one organization attempts to dominate the thinking, then I am sure that we can provide the programmer with effective tools for navigation.

Turing *It's Time to* Award *Reconsider Time* Lecture



When I was a student, there was no such thing as computer science and my interests were strictly mathematics. I graduated from Carlton College in 1958 with a BA in mathematics and went on to Princeton to do graduate work in mathematics. There, under the supervision of Harold W. Kuhn and the mentoring of Robert J. Aumann, I wrote a Ph.D. thesis on game theory entitled "Three-Person Cooperative Games without Side Payments" [8] and graduated in the fall of 1961.

In 1960, I was hired by Richard L. Shuey to work for the summer at General Electric's Research Laboratory in Schenectady, New York. Shuey was manager of a group called the "Information Studies Section" which included Juris Hartmanis and Philip M. Lewis II. This group was part of a larger group called "Electron Physics." During that summer, I worked with Hartmanis continuing some work he had started on the decomposition of sequential machines. As a result, the first Hartmanis-Stearns article [9] was produced. I was very impressed with the quality of the people at GE and the freedom they had to define their own research goals. Thus, when Shuey invited me to return in 1961 as a permanent employee, I jumped at the chance. There were a number of jokes made about my Ph.D. thesis since GE had just settled an antitrust suit for cooperating with two other companies to fix prices (no side payments involved!).

When I started at GE, it was strictly as a mathematician. The research laboratory did not have a computer and I had never used a computer. A few years later, the laboratory obtained a GE300 and installed a copy of the time-sharing system developed at Dartmouth. My first programming experiences were using Basic on this machine through a teletype interface. These experiences came after Hartmanis and I had worked out our theory of computational complexity. My transformation into a computer scientist was a matter of evolution rather than a conscious decision. I was pursuing the mathematical problems I found most interesting and found myself in the middle of computer science.

I continue to hold a simultaneous interest in both game theory and computer science. In fact, my first article with Hartmanis preceded my thesis work. You may wonder if these fields have anything in common. I think they do.

One commonality is that they were both started by John von Neumann. The beginning of game theory is clearly marked by the famous book by von Neumann and Morgenstern [13] and von Neumann with his "von Neumann stored program computer model" is also considered a founding father of computer science.

A second thing they have in common is the need to understand something very intangible yet very real. In the case of game theory, it is the nature of competition. In the case of computer science, it is the nature of computation. Both areas have required some completely new mathematics requiring the development of new mathematical models. My attraction to game theory began as an undergraduate one summer when I read [13] for recreation and saw how much attention was given to discussion about the choices of model.

To me, the most interesting mathematics is that which tells us something about the real world. Thus, the first question we as mathematicians and computer scientists should ask is "how well do our models reflect the salient features of the objects or situations we wish to describe?" The significance of a result depends more on the information it conveys rather than on the complexity of its proof. Garbage-in/garbage-out applies to theory as well as software.

Complexity

The complexity work with Hartmanis appeared in both a conference version [5] and a journal version [6].¹ The conference version was given at the Fifth Annual Symposium on Switching Circuit Theory and Logical Design which has undergone a couple of name changes and is now known as FOCS (Foundations of Computer Science). Both versions of the article had the phrase "computational complexity" in their title, the first time this phrase was used. Thus, we were the first to call what we were doing "computational complexity."

Although the conference version appeared before the journal version, the text of the conference version was written later and referred to itself as an "update" on the journal version. One update was inclusion of a reference to Blum's Ph.D. thesis from MIT which was to appear as [1]. This work, developed independently of ours, provided a more abstract view of complexity classes. This was a very significant contribution which should also be credited with inspiring the early interest and rapid growth of the complexity field.

In [6], we developed our theory on input-less multitape Turing machines which produced an infinite sequence of zeros and ones. This was essentially the same model used by Yamada in his work about "real-time countable functions" [14]. However, the now familiar language recognition model was rapidly ascending to the prominent position in automata theory it holds today. Recognizers are a valuable model because, despite their simplistic yes/no outputs, they are already sufficient to discuss most computational issues. Furthermore, they are ideally suited for the study of nondeterminism. In the update, we discussed the application of our methods to this model.

A third update was to discuss the implication of using the Hennie-Stearns two-tape $n \log n$ simulation [7] instead of the Hartmanis-Stearns one-tape n^2 simulation. (Fred Hennie worked at GE from time to time as a visitor.) The implication was a sharper, sufficient condition for distinguishing complexity classes.

The primary contribution of our work was the use of deterministic time to define complexity classes. Using modern notation and the language recognition model, our definition was this:

DEFINITION 1. $DTIME(T(n))$ is the set of all languages L for which there is a multitape Turing machine such that the machine

1. answers the question "does input w belong to language L ?" and
2. answers the question in at most $T(|w|)$ moves where $|w|$ is the length of input w .

In other words, a problem can be placed in the class $DTIME(T(n))$ by providing a program which answers the corresponding membership question in $T(n)$ steps where n is the size of the input. The most common objective in the analysis of algorithms is to place an upper bound on the running time of an algorithm. In terms of the time com-

¹There was also Research Laboratory report dated April, 1963.

plexity model, this corresponds to placing the problem solved by the algorithm into a complexity class.

In a sense, these complexity classes should be called "easiness classes" since expressing the idea that a problem is inherently hard (ie., establishing a lower bound) is achieved by showing that a problem does *not* belong to a certain time class.

One of the first things we proved was a "speed-up theorem":

THEOREM 1.

$$DTIME(T(n)) = DTIME(c \cdot T(n))$$

for all $c > 0$.²

In other words, it is $O(T(n))$ and not $T(n)$ which is important. This is a property we should expect from a proper model since there is no meaningful way to relate the number of transitions of an automaton to actual time measured in seconds and since the time in seconds to perform an algorithm will vary with computer technology. It was nice to see this property as a consequence of our definition. Of course constant factors are important in practice and not all $O(T(n))$ programs are equally good, but we do not expect to discriminate between such programs at an automata theoretic level.

The central result of our work was:

THEOREM 2. If

$$\lim_{n \rightarrow \infty} \frac{T(n) \cdot \log(T(n))}{U(n)} = 0$$

then $DTIME(U(n))$ contains a language which is not in $DTIME(T(n))$.³

This result implies that there really is a time hierarchy and that small changes in the time function give different classes. For example, there are problems that can be solved in $O(n^2)$ time that cannot be solved in $O(n^{2-\epsilon})$ time for any $\epsilon > 0$. Therefore, certain problems have an inherent complexity that cannot be circumvented by clever programming.

One property our model did not have was "machine independence." That is, if the complexity classes were defined with respect to some enhanced Turing machine, say with two-dimensional tapes, the classes would be somewhat different. Our article investigated several such enhancements and found that the time differences were related by low-degree polynomials. A complexity concept is now considered "machine independent" if it does not change with models polynomially related in time to Turing machines.

A short time later, Hartmanis and I looked at "tape" complexity (now known as space complexity) with Phil Lewis [10]. This included an innovation at the model level. We defined our space complexity with respect to the

amount of scratch tape used (ie., read-only input tapes would not be counted in the measurement). This enabled sublinear complexity classes all the way down to $\log n$ space and in some cases even to $\log \log n$ space.

Hardness Concepts

Although the time and space complexity classes form hierarchies and provided concepts for discussing hardness, there were no obvious techniques for taking a particular problem and showing it was not easy. In other words, it was (and still is) hard to show that there is no good method at all for solving a particular problem. Just because I have a clever method of solving a problem in say $O(T(n))$ time, how do I know that there does not exist an $O(T'(n))$ algorithm where $T'(n)$ is much smaller than $T(n)$?

The situation improved considerably when Cook introduced the concepts we now call NP-hardness and NP-completeness [2]. The overall idea was to relate the hardness of a particular problem to the hardness of some set of seemingly difficult problems. This would be done by showing that, if a good algorithm existed for the given problem, a good algorithm would exist for each problem in the set. In Cook's case, the set of seemingly difficult problems was the set of problems now called NP-complete and an algorithm would be considered "good" if it took only polynomial time. The hardness concept itself is now called NP-completeness.

The NP-complete problems seem to be very difficult because, by definition, if any of them can be solved in polynomial time, so can any problem which has a nondeterministic polynomial algorithm. In fact, these problems seem to require exponential time. The standard way of proving a problem X is NP-complete involves taking a problem Y already known to NP-complete and demonstrating that some polynomial time reduction can be used to convert any instance of Y into an instance of X having the same answer. In effect, the reduction implies that any method of solving X can be used to solve Y equally efficiently and so X cannot be easier than Y . Since we believe Y to be hard, we also believe X is hard. Thus, NP-hardness is accepted as good evidence that a problem requires exponential time.

To make this plan work, it is necessary to have some problems known to be NP-complete. Cook's article started things off by showing that the problem known as SAT, namely the satisfiability problem for CNF Boolean formulas, is NP-complete. This was quickly followed by the work by Karp [4] showing that many combinatorial problems of practical interest are also NP-complete. Soon hundreds of practical problems were shown to be NP-complete, many of which are found in [3].

Another hardness concept soon appeared, namely PSPACE-hardness, based on the concept of PSPACE-completeness. PSPACE-hard problems seem very difficult since if any of them can be solved in polynomial time, all problems in polynomial space can be solved in polynomial time. PSPACE-complete problems also seem to require exponential time. As with NP-completeness, the standard method of proving PSPACE-completeness involves poly-

²For purposes of this talk, I am somewhat simplifying theorem statements.

³Originally without [7], we had $[T(n)]^2$ instead of $T(n) \log T(n)$. Also $U(n)$ must be "time constructible."

nomial time reductions, this time from PSPACE-complete problems. PSPACE-hardness is stronger evidence of hardness than NP-hardness since PSPACE-hard problems might still require exponential time even if it unexpectedly turns out that the NP-complete problems can be solved in polynomial time. One of the first problems shown to be PSPACE-complete is QSAT, the problem of deciding if a quantified CNF formula is true [12].

Hardness concepts have proven to be very useful in classifying a variety of problems of practical interest and have contributed greatly to our understanding about the real world of computing. The application of these ideas has been so successful that we sometimes overlook their limitations or forget what they really mean. Some of these limitations will be pointed out here and then we will see that, from the viewpoint of deterministic time, there is a lot more to be learned about the time complexity of such problems.

Although PSPACE-completeness is stronger evidence of hardness than NP-completeness, there is no reason to believe that PSPACE-complete problems are harder in the sense that they require more time. Consider for example SAT which is NP-complete and QSAT which is PSPACE-complete. The best algorithms known for these problems are practically identical. Both involve considering all $\Theta(2^n)$ assignments to the variables and both use $\Theta(n)$ space. We are inclined to believe that SAT cannot be solved more quickly and therefore are inclined to believe that both SAT and QSAT have the same complexity.

When we say that NP-complete problems seem to take "exponential time," we do not mean 2^n . "Exponential" in this case means 2^{n^ϵ} for some $\epsilon > 0$. Thus 2^n , $2^{\sqrt{n}}$, $2^{\sqrt[3]{n}}$, and even $2^{\sqrt[10]{n}}$ are exponential. In fact, the classes $DTIME(2^n)$ contain PSPACE-complete and NP-complete problems for all $\epsilon > 0$. Time $\Theta(2^{n^\epsilon})$ may not be impossibly large when ϵ is small. For example when $n = 31,991$, $2^{\sqrt[3]{n}}$ operations can be performed in an hour on a 1 mips computer and $2^{\sqrt[10]{n}}$ is only about 10615.

Thus, it is conceivable that some of these "hard" problems can be easily solved for all inputs of significant size.

A Time-Based Perspective

In order to facilitate further discussion, consider the following time-based concept of *power index* that Harry B. Hunt III and I have developed [11]:

DEFINITION 2. *The power index of a problem L is the greatest lower bound on the set*

$$\{r \mid L \in DTIME(2^r)\}$$

if the set is nonempty and ∞ if the set is empty.

This concept has several attractive properties:

1. Every problem has a power index because every nonempty set of nonnegative reals has a greatest lower bound.
2. For every rational r , there is a problem with power index r . (This follows from Theorem 2.)
3. The power index concept is "machine independent."

Problems with polynomial algorithms (ie., problems in P) all have power index zero. Problems with an exponential time bound (ie., problems in EXPTIME) all have finite power indices.

If any NP-complete problem has a power index greater than zero, then all do and $P \neq NP$. This is the same for PSPACE-complete problems. Therefore, proving that an NP-complete problem has nonzero power index would prove $P \neq NP$ and proving that a PSPACE-complete problem has nonzero power index would prove $P \neq PSPACE$. Thus we expect difficulties in establishing the power indices of such problems. However, as with "evidence of hardness," we can study "evidence of power index" by using reductions. To do this, we must pay attention to reduction size defined as follows:

DEFINITION 3. *A reduction R is of size $s(n)$ if and only if the length of output $R(w)$ is $\Theta(s(n))$.*

Relationships between power indices can be established by the following:

THEOREM 3. *If the power index of L_1 is r and L_1 is reducible to L_2 by a polynomial time reduction of size n^s , then the power index of L_2 is at least r/s .*

Notice that the larger the size of the reduction (ie., the larger the degree s) the weaker the lower bound r/s . Just knowing that a reduction is polynomial conveys no information at all about power index.

To put Theorem 3 in perspective, consider SAT and CLIQUE. The standard reductions from SAT to CLIQUE [3, 4] have n^2 size and no better reduction is known. Thus, the strongest conclusion made from the theorem is that the power index of CLIQUE is at least one-half that of SAT. This actually fits the known facts quite well since the best-known algorithm for SAT takes $2^{\Theta(n)}$ time and the best algorithm for CLIQUE uses only $2^{\Theta(\sqrt{n})}$ time.⁴ If we could find a linear-sized reduction from SAT to CLIQUE, we would then have a $2^{\Theta(\sqrt{n})}$ time algorithm for SAT!

From the viewpoint of deterministic time, all NP-complete problems are not equally hard. By paying close attention to the size of reductions, sharper comparisons can be made between the complexities of various NP-complete problems. One way to assess our understanding is to ask the following question: Assuming that SAT does require $2^{\Theta(n)}$ time (ie., assuming the power index of SAT is one) what can be inferred about the power indices of other NP-complete problems?

In many cases, we have linear reductions from SAT to other problems solvable in $2^{\Theta(n)}$ time, and the answer to this question is that these also must have power index one. However, there are also many problems where the best known reductions are not linear and these problems are potentially much easier. In most cases, it is an open question if this potential is somehow achievable or if smaller reductions exist.

On a more theoretical level, power indices remind us

⁴The easier algorithm for CLIQUE is due to the facts that a clique of size k has $O(k^2)$ edges and a problem instance of length n has no more than n edges. The number of nodes in the largest clique is therefore at most $O(\sqrt{n})$ and an exhaustive search can be restricted accordingly.

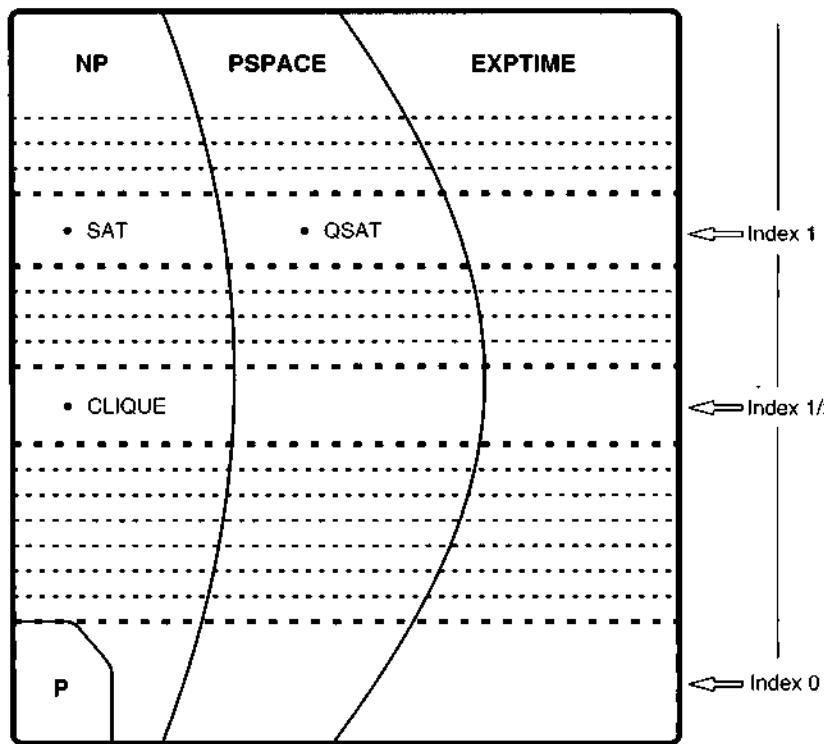


Figure 1.
EXPSPACE
subdivided
by power
index

that, to the best of our knowledge, the time hierarchy is largely orthogonal to the classes associated with evidence of hardness. Our best guess is that the world looks similar to the one shown in Figure 1. This figure shows the set EXPTIME carved up into classes of equal power index. These classes also carve up the sets PSPACE and NP. Both SAT and QSAT are shown, as conjectured, having power index one. Below that, CLIQUE is shown as conjectured with power index one-half and P is shown contained in the power index zero problems.

Our observations relating deterministic time to evidence of hardness can be summarized as follows:

- All NP-complete problems are not equally hard.
- All PSPACE-complete problems are not equally hard.
- A PSPACE-complete problem can be easier than an NP-complete problem.
- Even if SAT does require $2^{\Theta(n)}$ time, the possibility remains that many NP-complete problems of practical interest may require a lot less time. ■

References

1. Blum, M. A machine-independent theory of the complexity of recursive functions. *J. ACM* 14, 4 (Apr. 1967), 322–336.
2. Cook, S.A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, Shaker Heights, Ohio, 1971, pp. 151–158.
3. Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
4. Karp, R.M. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, eds., *Complexity of Computer Computations*, Plenum, N.Y. 1972, pp. 85–103.
5. Hartmanis, J. and Stearns, R.E. Computational complexity of recursive sequences. In *Proceedings of the Fifth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, Princeton, N.J., 1964, pp. 82–90.
6. Hartmanis, J. and Stearns, R.E. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.* 117, (May 1965), 285–305.
7. Hennie, F.C. and Stearns, R.E. Two-tape simulation of multi-tape turing machines. *J. of ACM*, 13, 10 (Oct. 1966), pp. 533–546.
8. Stearns, R.E. Three-person cooperative games without side payments. In M. Dresher, L.S. Shapley, and A.W. Tucker, eds., *Advances in Game Theory*, Annals of Mathematics Studies #52, Princeton University Press 1964, pp. 307–406.
9. Stearns, R.E. and Hartmanis, J. On the state assignment problem for sequential machines II. *IRE Trans. Electr. Comput., EC-10*, (Dec. 1961), 593–603.
10. Stearns, R.E., Hartmanis, J., and Lewis II, P.M. Hierarchies of memory limited computations. In *Proceedings of the Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, Ann Arbor, Mich. 1965, pp. 179–190.
11. Stearns, R.E. and Hunt III, H.B. Power indices and easier hard problems. *Math. Syst. Theory* 23 (1990), 209–225.
12. Stockmeyer, L.J. and Meyer, A.R. Word problems requiring exponential time. In *Proceedings of Fifth Annual ACM Symposium on the Theory of Computing*, Austin, Tex., 1973, pp. 1–9.
13. von Neumann, J. and Morgenstern, O. *Theory of Games and Economic Behavior*. Princeton, N.J., 1944.
14. Yamada, H. Real-time computation and recursive functions not real-time computable. *IRE Trans. Electr. Comput., EC-11*, (1962), 753–760.

About the Author:

RICHARD EDWIN STEARNS is a professor of computer science at the University at Albany, State University of New York. His current research interests include the structure of problem instances and the implications of such structure for complexity. **Author's Present Address:** University at Albany—SUNY, Computer Science Dept., Albany, NY 12222; email: res@cs.albany.edu

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/94/1000 \$3.50

The 1980 ACM Turing Award Lecture

Delivered at ACM '80, Nashville, Tennessee, October 27, 1980



C.A.R. Hoare

The 1980 ACM Turing Award was presented to Charles Antony Richard Hoare, Professor of Computation at the University of Oxford, England, by Walter Carlson, Chairman of the Awards committee, at the ACM Annual Conference in Nashville, Tennessee, October 27, 1980.

Professor Hoare was selected by the General Technical Achievement Award Committee for his fundamental contributions to the definition and design of programming languages. His work is characterized by an unusual combination of insight, originality, elegance, and impact. He is best known for his work on axiomatic definitions of programming languages through the use of techniques popularly referred to as axiomatic semantics. He developed ingenious algorithms such as Quicksort and was responsible for inventing and promulgating advanced data structuring techniques in scientific programming languages. He has also made important contributions to operating systems through the study of monitors. His most recent work is on communicating sequential processes.

Prior to his appointment to the University of Oxford in 1977, Professor Hoare was Professor of Computer Science at The Queen's University in Belfast, Ireland from 1968 to 1977 and was a Visiting Professor at Stanford University in 1973. From 1960

to 1968 he held a number of positions with Elliot Brothers, Ltd., England.

Professor Hoare has published extensively and is on the editorial boards of a number of the world's foremost computer science journals. In 1973 he received the ACM Programming Systems and Languages Paper Award. Professor Hoare became a Distinguished Fellow of the British Computer Society in 1978 and was awarded the degree of Doctor of Science *Honoris Causa* by the University of Southern California in 1979.

The Turing Award is the Association for Computing Machinery's highest award for technical contributions to the computing community. It is presented each year in commemoration of Dr. A. M. Turing, an English mathematician who made many important contributions to the computing sciences.

The Emperor's Old Clothes

Charles Antony Richard Hoare
Oxford University, England

The author recounts his experiences in the implementation, design, and standardization of computer programming languages, and issues a warning for the future.

Key Words and Phrases: programming languages, history of programming languages, lessons for the future

CR Categories: 1.2, 2.11, 4.2

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Author's present address: C. A. R. Hoare, 45 Banbury Road, Oxford OX2 6PE, England.
© 1981 ACM 0001-0782/81/0200-0075 \$00.75.

My first and most pleasant duty in this lecture is to express my profound gratitude to the Association for Computing Machinery for the great honor which they have bestowed on me and for this opportunity to address you on a topic of my choice. What a difficult choice it is! My scientific achievements, so amply recognized by this award, have already been amply described in the scientific literature. Instead of repeating the abstruse technicalities of my trade, I would like to talk informally about myself, my personal experiences, my hopes and fears, my modest successes, and my rather less modest failures. I have learned more from my failures than can ever be revealed in the cold print of a scientific article and now I would like you to learn from them, too. Besides, failures

are much more fun to hear about afterwards; they are not so funny at the time.

I start my story in August 1960, when I became a programmer with a small computer manufacturer, a division of Elliott Brothers (London) Ltd., where in the next eight years I was to receive my primary education in computer science. My first task was to implement for the new Elliot 803 computer, a library subroutine for a new fast method of internal sorting just invented by Shell. I greatly enjoyed the challenge of maximizing efficiency in the simple decimal-addressed machine code of those days. My boss and tutor, Pat Shackleton, was very pleased with my completed program. I then said timidly that I thought I had invented a sorting method that would usually run faster than SHELLSORT, without taking much extra store. He bet me sixpence that I had not. Although my method was very difficult to explain, he finally agreed that I had won my bet.

I wrote several other tightly coded library subroutines but after six months I was given a much more important task—that of designing a new advanced high level programming language for the company's next computer, the Elliott 503, which was to have the same instruction code as the existing 803 but run sixty times faster. In spite of my education in classical languages, this was a task for which I was even less qualified than those who undertake it today. By great good fortune there came into my hands a copy of the Report on the International Algorithmic Language ALGOL 60. Of course, this language was obviously too complicated for our customers. How could they ever understand all those begins and ends when even our salesmen couldn't?

Around Easter 1961, a course on ALGOL 60 was offered in Brighton, England, with Peter Naur, Edsger W. Dijkstra, and Peter Landin as tutors. I attended this course with my colleague in the language project, Jill Pym, our divisional Technical Manager, Roger Cook, and our Sales Manager, Paul King. It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who included recursion in their language and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.

After the ALGOL course in Brighton, Roger Cook was driving me and my colleagues back to London when he suddenly asked, "Instead of designing a new language, why don't we just implement ALGOL 60?" We all instantly agreed—in retrospect, a very lucky decision for me. But we knew we did not have the skill or experience at that time to implement the whole language, so I was commissioned to design a modest subset. In that design I

adopted certain basic principles which I believe to be as valid today as they were then.

(1) The first principle was *security*: The principle that every syntactically incorrect program should be rejected by the compiler and that every syntactically correct program should give a result or an error message that was predictable and comprehensible in terms of the source language program itself. Thus no core dumps should ever be necessary. It was logically impossible for any source language program to cause the computer to run wild, either at compile time or at run time. A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency on production runs. Unanimously, they urged us not to—they already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.

(2) The second principle in the design of the implementation was *brevity of the object code produced by the compiler and compactness of run time working data*. There was a clear reason for this: The size of main storage on any computer is limited and its extension involves delay and expense. A program exceeding the limit, even by one word, is impossible to run, especially since many of our customers did not intend to purchase backing stores.

This principle of compactness of object code is even more valid today, when processors are trivially cheap in comparison with the amounts of main store they can address, and backing stores are comparatively even more expensive and slower by many orders of magnitude. If as a result of care taken in implementation the available hardware remains more powerful than may seem necessary for a particular application, the applications programmer can nearly always take advantage of the extra capacity to increase the quality of his program, its simplicity, its ruggedness, and its reliability.

(3) The third principle of our design was that *the entry and exit conventions for procedures and functions should be as compact and efficient as for tightly coded machine-code subroutines*. I reasoned that procedures are one of the most powerful features of a high level language, in that they both simplify the programming task and shorten the object code. Thus there must be no impediment to their frequent use.

(4) The fourth principle was that *the compiler should use only a single pass*. The compiler was structured as a collection of mutually recursive procedures, each capable

of analyzing and translating a major syntactic unit of the language—a statement, an expression, a declaration, and so on. It was designed and documented in ALGOL 60, and then coded into decimal machine code using an explicit stack for recursion. Without the ALGOL 60 concept of recursion, at that time highly controversial, we could not have written this compiler at all.

I can still recommend single-pass top-down recursive descent both as an implementation method and as a design principle for a programming language. First, we certainly want programs to be read by *people* and people prefer to read things once in a single pass. Second, for the user of a time-sharing or personal computer system, the interval between typing in a program (or amendment) and starting to run that program is wholly unproductive. It can be minimized by the high speed of a single pass compiler. Finally, to structure a compiler according to the syntax of its input language makes a great contribution to ensuring its correctness. Unless we have absolute confidence in this, we can never have confidence in the results of any of our programs.

To observe these four principles, I selected a rather small subset of ALGOL 60. As the design and implementation progressed, I gradually discovered methods of relaxing the restrictions without compromising any of the principles. So in the end we were able to implement nearly the full power of the whole language, including even recursion, although several features were removed and others were restricted.

In the middle of 1963, primarily as a result of the work of Jill Pym and Jeff Hillmore, the first version of our compiler was delivered. After a few months we began to wonder whether anyone was using the language or taking any notice of our occasional reissue, incorporating improved operating methods. Only when a customer had a complaint did he contact us and many of them had no complaints. Our customers have now moved on to more modern computers and more fashionable languages but many have told me of their fond memories of the Elliott ALGOL System and the fondness is not due just to nostalgia, but to the efficiency, reliability, and convenience of that early simple ALGOL System.

As a result of this work on ALGOL, in August 1962, I was invited to serve on the new Working Group 2.1 of IFIP, charged with responsibility for maintenance and development of ALGOL. The group's first main task was to design a subset of the language which would remove some of its less successful features. Even in those days and even with such a simple language, we recognized that a subset could be an improvement on the original. I greatly welcomed the chance of meeting and hearing the wisdom of many of the original language designers. I was astonished and dismayed at the heat and even rancor of their discussions. Apparently the original design of ALGOL 60 had not proceeded in that spirit of

dispassionate search for truth which the quality of the language had led me to suppose.

In order to provide relief from the tedious and argumentative task of designing a subset, the working group allocated one afternoon to discussing the features that should be incorporated in the next design of the language. Each member was invited to suggest the improvement he considered most important. On October 11, 1963, my suggestion was to pass on a request of our customers to relax the ALGOL 60 rule of compulsory declaration of variable names and adopt some reasonable default convention such as that of FORTRAN. I was astonished by the polite but firm rejection of this seemingly innocent suggestion: It was pointed out that the redundancy of ALGOL 60 was the best protection against programming and coding errors which could be extremely expensive to detect in a running program and even more expensive not to. The story of the Mariner space rocket to Venus, lost because of the lack of compulsory declarations in FORTRAN, was not to be published until later. I was eventually persuaded of the need to design programming notations so as to maximize the number of errors which cannot be made, or if made, can be reliably detected at compile time. Perhaps this would make the text of programs longer. Never mind! Wouldn't you be delighted if your Fairy Godmother offered to wave her wand over your program to remove all its errors and only made the condition that you should write out and key in your whole program three times! The way to shorten programs is to use procedures, not to omit vital declarative information.

Among the other proposals for the development of a new ALGOL was that the switch declaration of ALGOL 60 should be replaced by a more general feature, namely an array of label-valued variables and that a program should be able to change the values of these variables by assignment. I was very much opposed to this idea, similar to the assigned, GO TO of FORTRAN, because I had found a surprising number of tricky problems in the implementation of even the simple labels and switches of ALGOL 60. I could see even more problems in the new feature including that of jumping back into a block after it had been exited. I was also beginning to suspect that programs that used a lot of labels were more difficult to understand and get correct and that programs that assigned new values to label variables would be even more difficult still.

It occurred to me that the appropriate notation to replace the ALGOL 60 switch should be based on that of the conditional expression of ALGOL 60, which selects between two alternative actions according to the value of a Boolean expression. So I suggested the notation for a "case expression" which selects between any number of alternatives according to the value of an integer expression. That was my second language design proposal. I am still most proud of it, because it raises essentially no problems either for the implementor, the

programmer, or the reader of a program. Now, after more than fifteen years, there is the prospect of international standardization of a language incorporating this notation—a remarkably *short* interval compared with other branches of engineering.

Back again to my work at Elliott's. After the unexpected success of our ALGOL Compiler, our thoughts turned to a more ambitious project: To provide a range of operating system software for larger configurations of the 503 computer, with card readers, line printers, magnetic tapes, and even a core backing store which was twice as cheap and twice as large as main store, but fifteen times slower. This was to be known as the Elliott 503 Mark II software system.

It comprised:

- (1) An assembler for a symbolic assembly language in which all the rest of the software was to be written.
- (2) A scheme for automatic administration of code and data overlays, either from magnetic tape or from core backing store. This was to be used by the rest of the software.
- (3) A scheme for automatic buffering of all input and output on any available peripheral device,—again, to be used by all the other software.
- (4) A filing system on magnetic tape with facilities for editing and job control.
- (5) A completely new implementation of ALGOL 60, which removed all the nonstandard restrictions which we had imposed on our first implementation.
- (6) A compiler for FORTRAN as it was then.

I wrote documents which described the relevant concepts and facilities and we sent them to existing and prospective customers. Work started with a team of fifteen programmers and the deadline for delivery was set some eighteen months ahead in March 1965. After initiating the design of the Mark II software, I was suddenly promoted to the dizzying rank of Assistant Chief Engineer, responsible for advanced development and design of the company's products, both hardware and software.

Although I was still managerially responsible for the 503 Mark II software, I gave it less attention than the company's new products and almost failed to notice when the deadline for its delivery passed without event. The programmers revised their implementation schedules and a new delivery date was set some three months ahead in June 1965. Needless to say, that day also passed without event. By this time, our customers were getting angry and my managers instructed me to take personal charge of the project. I asked the senior programmers once again to draw up revised schedules, which again showed that the software could be delivered within another three months. I desperately wanted to believe it but I just could not. I disregarded the schedules and began to dig more deeply into the project.

It turned out that we had failed to make any overall

plans for the allocation of our most limited resource—main storage. Each programmer expected this to be done automatically, either by the symbolic assembler or by the automatic overlay scheme. Even worse, we had failed to simply count the space used by our own software which was already filling the main store of the computer, leaving no space for our customers to run *their* programs. Hardware address length limitations prohibited adding more main storage.

Clearly, the original specifications of the software could not be met and had to be drastically curtailed. Experienced programmers and even managers were called back from other projects. We decided to concentrate first on delivery of the new compiler for ALGOL 60, which careful calculation showed would take another four months. I impressed upon all the programmers involved that this was no longer just a prediction; it was a promise; if they found they were not meeting their promise, it was their personal responsibility to find ways and means of making good.

The programmers responded magnificently to the challenge. They worked nights and days to ensure completion of all those items of software which were needed by the ALGOL compiler. To our delight, they met the scheduled delivery date; it was the first major item of working software produced by the company over a period of two years.

Our delight was short-lived; the compiler could not be delivered. Its speed of compilation was only two characters per second which compared unfavorably with the existing version of the compiler operating at about a thousand characters per second. We soon identified the cause of the problem: It was thrashing between the main store and the extension core backing store which was fifteen times slower. It was easy to make some simple improvements, and within a week we had doubled the speed of compilation—to four characters per second. In the next two weeks of investigation and reprogramming, the speed was doubled again—to eight characters per second. We could see ways in which within a month this could be still further improved; but the amount of reprogramming required was increasing and its effectiveness was decreasing; there was an awful long way to go. The alternative of increasing the size of the main store so frequently adopted in later failures of this kind was prohibited by hardware addressing limitations.

There was no escape: The entire Elliott 503 Mark II software project had to be abandoned, and with it, over thirty man-years of programming effort, equivalent to nearly one man's active working life, and I was responsible, both as designer and as manager, for wasting it.

A meeting of all our 503 customers was called and Roger Cook, who was then manager of the computing division, explained to them that not a single word of the long-promised software would ever be delivered to them. He adopted a very quiet tone of delivery, which ensured that none of the customers could interrupt, murmur in

the background, or even shuffle in their seats. I admired but could not share his calm. Over lunch our customers were kind to try to comfort me. They had realized long ago that software to the original specification could never have been delivered, and even if it had been, they would not have known how to use its sophisticated features, and anyway many such large projects get cancelled before delivery. In retrospect, I believe our customers were fortunate that hardware limitations had protected them from the arbitrary excesses of our software designs. In the present day, users of microprocessors benefit from a similar protection—but not for much longer.

At that time I was reading the early documents describing the concepts and features of the newly announced OS 360, and of a new time-sharing project called Multics. These were far more comprehensive, elaborate, and sophisticated than anything I had imagined, even in the first version of the 503 Mark II software. Clearly IBM and MIT must be possessed of some secret of successful software design and implementation whose nature I could not even begin to guess at. It was only later that they realized they could not either.

So I still could not see how I had brought such a great misfortune upon my company. At the time I was convinced that my managers were planning to dismiss me. But no, they were intending a far more severe punishment. "O.K. Tony," they said. "You got us into this mess and now you're going to get us out." "But I don't know how," I protested, but their reply was simple. "Well then, you'll have to find out." They even expressed confidence that I could do so. I did not share their confidence. I was tempted to resign. It was the luckiest of all my lucky escapes that I did not.

Of course, the company did everything they could to help me. They took away my responsibility for hardware design and reduced the size of my programming teams. Each of my managers explained carefully his own theory of what had gone wrong and all the theories were different. At last, there breezed into my office the most senior manager of all, a general manager of our parent company, Andrew St. Johnston. I was surprised that he had even heard of me. "You know what went wrong?" he shouted—he always shouted—"You let your programmers do things which you yourself do not understand." I stared in astonishment. He was obviously out of touch with present day realities. How could one person ever understand the whole of a modern software product like the Elliott 503 Mark II software system?

I realized later that he was absolutely right; he had diagnosed the true cause of the problem and he had planted the seed of its later solution.

I still had a team of some forty programmers and we needed to retain the good will of customers for our new machine and even regain the confidence of the customers for our old one. But what should we actually plan to do when we knew only one thing—that all our previous plans had failed? I therefore called an all-day meeting of

our senior programmers on October 22, 1965, to thrash out the question between us. I still have the notes of that meeting. We first listed the recent major grievances of our customers: Cancellation of products, failure to meet deadlines, excessive size of software, "... not justified by the usefulness of the facilities provided," excessively slow programs, failure to take account of customer feedback; "Earlier attention paid to quite minor requests of our customers might have paid as great dividends of goodwill as the success of our most ambitious plans."

We then listed our own grievances: Lack of machine time for program testing, unpredictability of machine time, lack of suitable peripheral equipment, unreliability of the hardware even when available, dispersion of programming staff, lack of equipment for keypunching of programs, lack of firm hardware delivery dates, lack of technical writing effort for documentation, lack of software knowledge outside of the programming group, interference from higher managers who imposed decisions, "... without a full realization of the more intricate implications of the matter," and overoptimism in the face of pressure from customers and the Sales Department.

But we did not seek to excuse our failure by these grievances. For example, we admitted that it was the duty of programmers to educate their managers and other departments of the company by "... presenting the necessary information in a simple palatable form." The hope "... that deficiencies in original program specifications could be made up by the skill of a technical writing department..." was misguided; the design of a program and the design of its specification must be undertaken in parallel by the same person, and they must interact with each other. A lack of clarity in specification is one of the surest signs of a deficiency in the program it describes, and the two faults must be removed simultaneously before the project is embarked upon." I wish I had followed this advice in 1963; I wish we all would follow it today.

My notes of the proceedings of that day in October 1965 include a complete section devoted to failings within the software group; this section rivals the most abject self-abasement of a revisionist official in the Chinese cultural revolution. Our main failure was overambition. "The goals which we have attempted have obviously proved to be far beyond our grasp." There was also failure in prediction, in estimation of program size and speed, of effort required, in planning the coordination and interaction of programs, in providing an early warning that things were going wrong. There were faults in our control of program changes, documentation, liaison with other departments, with our management, and with our customers. We failed in giving clear and stable definitions of the responsibilities of individual programmers and project leaders,—Oh, need I go on? What was amazing was that a large team of highly intelligent programmers could labor so hard and so long on such

an unpromising project. You know, you shouldn't trust us intelligent programmers. We can think up such good arguments for convincing ourselves and each other of the utterly absurd. Especially don't believe us when we promise to repeat an earlier success, only bigger and better next time.

The last section of our inquiry into the failure dealt with the criteria of quality of software. "In the recent struggle to deliver any software at all, the first casualty has been consideration of the quality of the software delivered. The quality of software is measured by a number of totally incompatible criteria, which must be carefully balanced in the design and implementation of every program." We then made a list of no less than seventeen criteria which has been published in a guest editorial in Volume 2 of the journal, *Software Practice and Experience*.

How did we recover from the catastrophe? First, we classified our 503 customers into groups, according to the nature and size of the hardware configurations which they had bought—for example, those with magnetic tapes were all in one group. We assigned to each group of customers a small team of programmers and told the team leader to visit the customers to find out what they wanted; to select the easiest request to fulfil, and to make plans (but not promises) to implement it. In no case would we consider a request for a feature that would take more than three months to implement and deliver. The project leader would then have to convince me that the customers' request was reasonable, that the design of the new feature was appropriate, and that the plans and schedules for implementation were realistic. Above all, I did not allow anything to be done which I did not myself understand. It worked! The software requested began to be delivered on the promised dates. With an increase in our confidence and that of our customers, we were able to undertake fulfilling slightly more ambitious requests. Within a year we had recovered from the disaster. Within two years, we even had some moderately satisfied customers.

Thus we muddled through by common sense and compromise to something approaching success. But I was not satisfied. I did not see why the design and implementation of an operating system should be so much more difficult than that of a compiler. This is the reason why I have devoted my later research to problems of parallel programming and language constructs which would assist in clear structuring of operating systems—constructs such as monitors and communicating processes.

While I was working at Elliotts', I became very interested in techniques for formal definition of programming languages. At that time, Peter Landin and Christopher Strachey proposed to define a programming language in a simple functional notation, that specified the effect of each command on a mathematically defined abstract machine. I was not happy with this proposal

because I felt that such a definition must incorporate a number of fairly arbitrary representation decisions and would not be much simpler in principle than an implementation of the language for a real machine. As an alternative, I proposed that a programming language definition should be formalized as a set of axioms, describing the desired properties of programs written in the language. I felt that carefully formulated axioms would leave an implementation the necessary freedom to implement the language efficiently on different machines and enable the programmer to prove the correctness of his programs. But I did not see how to actually do it. I thought that it would need lengthy research to develop and apply the necessary techniques and that a university would be a better place to conduct such research than industry. So I applied for a chair in Computer Science at the Queen's University of Belfast where I was to spend nine happy and productive years. In October 1968, as I unpacked my papers in my new home in Belfast, I came across an obscure preprint of an article by Bob Floyd entitled, "Assigning Meanings to Programs." What a stroke of luck! At last I could see a way to achieve my hopes for my research. Thus I wrote my first paper on the axiomatic approach to computer programming, published in the *Communications of the ACM* in October 1969.

Just recently, I have discovered that an early advocate of the assertional method of program proving was none other than Alan Turing himself. On June 24, 1950 at a conference in Cambridge, he gave a short talk entitled, "Checking a Large Routine" which explains the idea with great clarity. "How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite *assertions* which can be checked individually, and from which the correctness of the whole program easily follows."

Consider the analogy of checking an addition. If the sum is given [just as a column of figures with the answer below] one must check the whole at one sitting. But if the totals for the various columns are given, [with the carries added in separately], the checker's work is much easier, being split up into the checking of the various assertions [that each column is correctly added] and the small addition [of the carries to the total]. This principle can be applied to the checking of a large routine but we will illustrate the method by means of a small routine viz. one to obtain n factorial without the use of a multiplier. Unfortunately there is no coding system sufficiently generally known to justify giving this routine in full, but a flow diagram will be sufficient for illustration. That brings me back to the main theme of my talk, the design of programming languages.

During the period, August 1962 to October 1966, I attended every meeting of the IFIP ALGOL working group. After completing our labors on the IFIP ALGOL subset, we started on the design of ALGOL X, the intended

successor to ALGOL 60. More suggestions for new features were made and in May 1965, Niklaus Wirth was commissioned to collate them into a single language design. I was delighted by his draft design which avoided all the known defects of ALGOL 60 and included several new features, all of which could be simply and efficiently implemented, and safely and conveniently used.

The description of the language was not yet complete. I worked hard on making suggestions for its improvement and so did many other members of our group. By the time of the next meeting in St. Pierre de Chartreuse, France in October 1965, we had a draft of an excellent and realistic language design which was published in June 1966 as "A Contribution to the Development of ALGOL", in the *Communications of the ACM*. It was implemented on the IBM 360 and given the title ALGOL w by its many happy users. It was not only a worthy successor of ALGOL 60, it was even a worthy predecessor of PASCAL.

At the same meeting, the ALGOL committee had placed before it, a short, incomplete and rather incomprehensible document, describing a different, more ambitious and, to me, a far less attractive language. I was astonished when the working group, consisting of all the best known international experts of programming languages, resolved to lay aside the commissioned draft on which we had all been working and swallow a line with such an unattractive bait.

This happened just one week after our inquest on the 503 Mark II software project. I gave desperate warnings against the obscurity, the complexity, and overambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are *no obvious* deficiencies.

The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature. It also requires a willingness to accept objectives which are limited by physical, logical, and technological constraints, and to accept a compromise when conflicting objectives cannot be met. No committee will ever do this until it is too late.

So it was with the ALGOL committee. Clearly the draft which it preferred was not yet perfect. So a new and final draft of the new ALGOL language design was promised in three months' time; it was to be submitted to the scrutiny of a subgroup of four members including myself. Three months came and went, without a word of the new draft. After six months, the subgroup met in the Netherlands. We had before us a longer and thicker document, full of errors corrected at the last minute, describing yet another but to me, equally unattractive language. Niklaus Wirth and I spent some time trying to get removed some of the deficiencies in the design and

in the description, but in vain. The completed final draft of the language was promised for the next meeting of the full ALGOL committee in three months time.

Three months came and went—not a word of the new draft appeared. After six months, in October 1966, the ALGOL working group met in Warsaw. It had before it an even longer and thicker document, full of errors corrected at the last minute, describing equally obscurely yet another different, and to me, equally unattractive language. The experts in the group could not see the defects of the design and they firmly resolved to adopt the draft, believing it would be completed in three months. In vain, I told them it would not. In vain, I urged them to remove some of the technical mistakes of the language, the predominance of references, the default type conversions. Far from wishing to simplify the language, the working group actually asked the authors to include even more complex features like overloading of operators and concurrency.

When any new language design project is nearing completion, there is always a mad rush to get new features added before standardization. The rush is mad indeed, because it leads into a trap from which there is no escape. A feature which is omitted can always be added later, when its design and its implications are well understood. A feature which is included before it is fully understood can never be removed later.

At last, in December 1968, in a mood of black depression, I attended the meeting in Munich at which our long-gestated monster was to come to birth and receive the name ALGOL 68. By this time, a number of other members of the group had become disillusioned, but too late: The committee was now packed with supporters of the language, which was sent up for promulgation by the higher committees of IFIP. The best we could do was to send with it a minority report, stating our considered view that, "... as a tool for the *reliable creation* of sophisticated programs, the language was a failure." This report was later suppressed by IFIP, an act which reminds me of the lines of Hilaire Belloc,

But scientists, who ought to know/Assure us that it must be so./
Oh, let us never, never doubt/What nobody is sure about.

I did not attend any further meetings of that working group. I am pleased to report that the group soon came to realize that there was something wrong with their language and with its description; they labored hard for six more years to produce a revised description of the language. It is a great improvement but I'm afraid, that in my view, it does not remove the basic technical flaws in the design, nor does it begin to address the problem of its overwhelming complexity.

Programmers are always surrounded by complexity; we cannot avoid it. Our applications are complex because we are ambitious to use our computers in ever more sophisticated ways. Programming is complex because of

the large number of conflicting objectives for each of our programming projects. If our basic tool, the language in which we design and code our programs, is also complicated, the language itself becomes part of the problem rather than part of its solution.

Now let me tell you about yet another overambitious language project. Between 1965 and 1970 I was a member and even chairman of the Technical Committee No. 10 of the European Computer Manufacturers Association. We were charged first with a watching brief and then with the standardization of a language to end all languages, designed to meet the needs of all computer applications, both commercial and scientific, by the greatest computer manufacturer of all time. I had studied with interest and amazement, even a touch of amusement, the four initial documents describing a language called NPL, which appeared between March 1 and November 30, 1964. Each was more ambitious and absurd than the last in its wishful speculations. Then the language began to be implemented and a new series of documents began to appear at six-monthly intervals, each describing the final frozen version of the language, under its final frozen name PL/I.

But to me, each revision of the document simply showed how far the initial *F*-level implementation had progressed. Those parts of the language that were not yet implemented were still described in free-flowing flowery prose giving promise of unalloyed delight. In the parts that *had* been implemented, the flowers had withered; they were choked by an undergrowth of explanatory footnotes, placing arbitrary and unpleasant restrictions on the use of each feature and loading upon a programmer the responsibility for controlling the complex and unexpected side-effects and interaction effects with all the other features of the language.

At last, March 11, 1968, the language description was nobly presented to the waiting world as a worthy candidate for standardization. But it was not. It had already undergone some seven thousand corrections and modifications at the hand of its original designers. Another twelve editions were needed before it was finally published as a standard in 1976. I fear that this was not because everybody concerned was satisfied with its design, but because they were thoroughly bored and disillusioned.

For as long as I was involved in this project, I urged that the language be simplified, if necessary by subsetting, so that the professional programmer would be able to understand it and be able to take responsibility for the correctness and cost-effectiveness of his programs. I urged that the dangerous features such as defaults and ON- conditions be removed. I knew that it would be impossible to write a wholly reliable compiler for a language of this complexity and impossible to write a wholly reliable program when the correctness of each part of the program depends on checking that every other part of the program has avoided all the traps and pitfalls of the language.

At first I hoped that such a technically unsound project would collapse but I soon realized it was doomed to success. Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. But there is one quality that cannot be purchased in this way—and that is reliability. The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay.

All this happened a long time ago. Can it be regarded as relevant in a conference dedicated to a preview of the Computer Age that lies ahead? It is my gravest fear that it can. The mistakes which have made in the last twenty years are being repeated today on an even grander scale. I refer to a language design project which has generated documents entitled *strawman*, *woodenman*, *tinman*, *ironman*, *steelman*, *green* and finally now ADA. This project has been initiated and sponsored by one of the world's most powerful organizations, the United States Department of Defense. Thus it is ensured of an influence and attention quite independent of its technical merits and its faults and deficiencies threaten us with far greater dangers. For none of the evidence we have so far can inspire confidence that this language has avoided any of the problems that have afflicted other complex language projects of the past.

I have been giving the best of my advice to this project since 1975. At first I was extremely hopeful. The original objectives of the language included reliability, readability of programs, formality of language definition, and even simplicity. Gradually these objectives have been sacrificed in favor of power, supposedly achieved by a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous. We relive the history of the design of the motor car. Gadgets and glitter prevail over fundamental concerns of safety and economy.

It is not too late! I believe that by careful pruning of the ADA language, it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic in use. The sponsors of the language have declared unequivocally, however, that there shall be no subsets. This is the strangest paradox of the whole strange project. If you want a language with no subsets, you must make it *small*.

You include only those features which you know to be needed for every single application of the language and which you know to be appropriate for every single hardware configuration on which the language is implemented. Then extensions can be specially designed where necessary for particular hardware devices and for particular applications. That is the great strength of PASCAL, that there are so few unnecessary features and almost no need for subsets. That is why the language is strong enough to support specialized extensions—Concurrent PASCAL for real time work, PASCAL PLUS for discrete event simulation, UCSD PASCAL for microprocessor work

stations. If only we could learn the right lessons from the successes of the past, we would not need to learn from our failures.

And so, the best of my advice to the originators and designers of ADA has been ignored. In this last resort, I appeal to you, representatives of the programming profession in the United States, and citizens concerned with the welfare and safety of your own country and of mankind: Do not allow this language in its present state to be used in applications where reliability is critical, i.e., nuclear power stations, cruise missiles, early warning systems, anti-ballistic missile defense systems. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our own cities. An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.

Let me not end on this somber note. To have our best advice ignored is the common fate of all who take on the role of consultant, ever since Cassandra pointed out the dangers of bringing a wooden horse within the walls of Troy. That reminds me of a story I used to hear in my childhood. As far as I recall, its title was:

The Emperor's Old Clothes

Many years ago, there was an Emperor who was so excessively fond of clothes that he spent all his money on dress. He did not trouble himself with soldiers, attend banquets, or give judgement in court. Of any other king or emperor one might say, "He is sitting in council," but it was always said of him, "The emperor is sitting in his wardrobe." And so he was. On one unfortunate occasion, he had been tricked into going forth naked to his chagrin and the glee of his subjects. He resolved never to leave his throne, and to avoid nakedness, he ordered that each of his many new suits of clothes should be simply draped on top of the old.

Time passed away merrily in the large town that was his capital. Ministers and courtiers, weavers and tailors, visitors and subjects, seamstresses and embroiderers, went in and out of the throne room about their various tasks, and they all exclaimed, "How magnificent is the attire of our Emperor."

One day the Emperor's oldest and most faithful Minister heard tell of a most distinguished tailor who taught at an ancient institute of higher stitchcraft, and who had developed a new art of abstract embroidery using stitches so refined that no one could tell whether they were actually there at all. "These must indeed be splendid stitches," thought the minister. "If we can but engage this tailor to advise us, we will bring the adornment of our Emperor to such heights of ostentation that all the world will acknowledge him as the greatest Emperor there has ever been."

So the honest old Minister engaged the master tailor at vast expense. The tailor was brought to the throne room where he made obeisance to the heap of fine clothes which now completely covered the throne. All the courtiers waited eagerly for his advice. Imagine their astonishment when his advice was not to add sophistication and more intricate embroidery to that which already existed, but rather to remove layers of the finery, and strive for simplicity and elegance in place of extravagant elaboration. "This tailor is not the expert that he claims," they muttered. "His wits have been addled by long contemplation in his ivory tower and he no longer understands the sartorial needs of a modern Emperor." The tailor argued loud and long for the good sense of his advice but could not make himself heard. Finally, he accepted his fee and returned to his ivory tower.

Never to this very day has the full truth of this story been told: That one fine morning, when the Emperor felt hot and bored, he extricated himself carefully from under his mountain of clothes and is now living happily as a swineherd in another story. The tailor is canonized as the patron saint of all consultants, because in spite of the enormous fees that he extracted, he was never able to convince his clients of his dawning realization that their clothes have no Emperor.

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX¹ swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write

programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE 1

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in the C³ programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

¹UNIX is a trademark of AT&T Bell Laboratories.

© 1984 0001-0782/84/0800-0761 75¢

```

char s[] = {
    '\v',
    '\0',
    '\n',
    '\r',
    '\t',
    '\v',
    '\n',
    '\r',
    '\t',
    '*',
    '\n',
    (213 lines deleted)
    0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
    int i;

    printf("char\ts[ ] = {\n");
    for(i=0; s[i]; i++)
        printf("\t%d, \n", s[i]);
    printf("%s", s);
}

```

Here are some simple transliterations to allow a non-C programmer to read this code.

- = assignment
- == equal to .EQ.
- != not equal to .NE.
- ++ increment
- 'x' single character constant
- "xxx" multiple character string
- %d format to convert to decimal
- %s format to convert to string
- \t tab character
- \n newline character

FIGURE 1.

STAGE II

The C compiler is written in C. What I am about to describe is one of many "chicken and egg" problems that arise when compilers are written in their own language. In this case, I will use a specific example from the C compiler.

C allows a string construct to specify an initialized character array. The individual characters in the string can be escaped to represent unprintable characters. For example,

"Hello world\n"

represents a string with the character "\n," representing the new line character.

Figure 2.1 is an idealization of the code in the C compiler that interprets the character escape sequence. This is an amazing piece of code. It "knows" in a completely portable way what character code is compiled for a new line in any character set. The act of knowing

then allows it to recompile itself, thus perpetuating the knowledge.

Suppose we wish to alter the C compiler to include the sequence "\v" to represent the vertical tab character. The extension to Figure 2.1 is obvious and is presented in Figure 2.2. We then recompile the C compiler, but we get a diagnostic. Obviously, since the binary version of the compiler does not know about "\v," the source is not legal C. We must "train" the compiler. After it "knows" what "\v" means, then our new change will become legal C. We look up on an ASCII chart that a vertical tab is decimal 11. We alter our source to look like Figure 2.3. Now the old compiler accepts the new source. We install the resulting binary as the new official C compiler and now we can write the portable version the way we had it in Figure 2.2.

This is a deep concept. It is as close to a "learning" program as I have seen. You simply tell it once, then you can use this self-referencing definition.

STAGE III

Again, in the C compiler, Figure 3.1 represents the high level control of the C compiler where the routine "com-

```

...
c = next( );
if(c != '\v')
    return(c);
c = next( );
if(c == '\v')
    return('\v');
if(c == '\n')
    return('\n');
...

```

FIGURE 2.2.

```

...
c = next( );
if(c != '\v')
    return(c);
c = next( );
if(c == '\v')
    return('\v');
if(c == '\n')
    return('\n');
if(c == 'v')
    return('v');
...

```

FIGURE 2.1.

```

...
c = next( );
if(c != '\v')
    return(c);
c = next( );
if(c == '\v')
    return('\v');
if(c == '\n')
    return('\n');
if(c == 'v')
    return('v');
return(11);
...

```

FIGURE 2.3.

pile" is called to compile the next line of source. Figure 3.2 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to compile the login command, I could log into that system as any user.

Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions.

The final step is represented in Figure 3.3. This simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. The replacement code is a Stage I self-reproducing program that inserts both Trojan horses into the compiler. This requires a learning phase as in the Stage II example. First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere.

```
compile(s)
char *s;
{
    ...
}
```

FIGURE 3.1.

```
compile(s)
char *s;
{
    if(match(s, "pattern")) {
        compile("bug");
        return;
    }
    ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
    if(match(s, "pattern1")) {
        compile ("bug1");
        return;
    }
    if(match(s, "pattern 2")) {
        compile ("bug 2");
        return;
    }
    ...
}
```

FIGURE 3.3.

MORAL

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well-installed microcode bug will be almost impossible to detect.

After trying to convince you that I cannot be trusted, I wish to moralize. I would like to criticize the press in its handling of the "hackers," the 414 gang, the Dalton gang, etc. The acts performed by these kids are vandalism at best and probably trespass and theft at worst. It is only the inadequacy of the criminal code that saves the hackers from very serious prosecution. The companies that are vulnerable to this activity, (and most large companies are very vulnerable) are pressing hard to update the criminal code. Unauthorized access to computer systems is already a serious crime in a few states and is currently being addressed in many more state legislatures as well as Congress.

There is an explosive situation brewing. On the one hand, the press, television, and movies make heroes of vandals by calling them whiz kids. On the other hand, the acts performed by these kids will soon be punishable by years in prison.

I have watched kids testifying before Congress. It is clear that they are completely unaware of the seriousness of their acts. There is obviously a cultural gap. The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked. The press must learn that misguided use of a computer is no more amazing than drunk driving of an automobile.

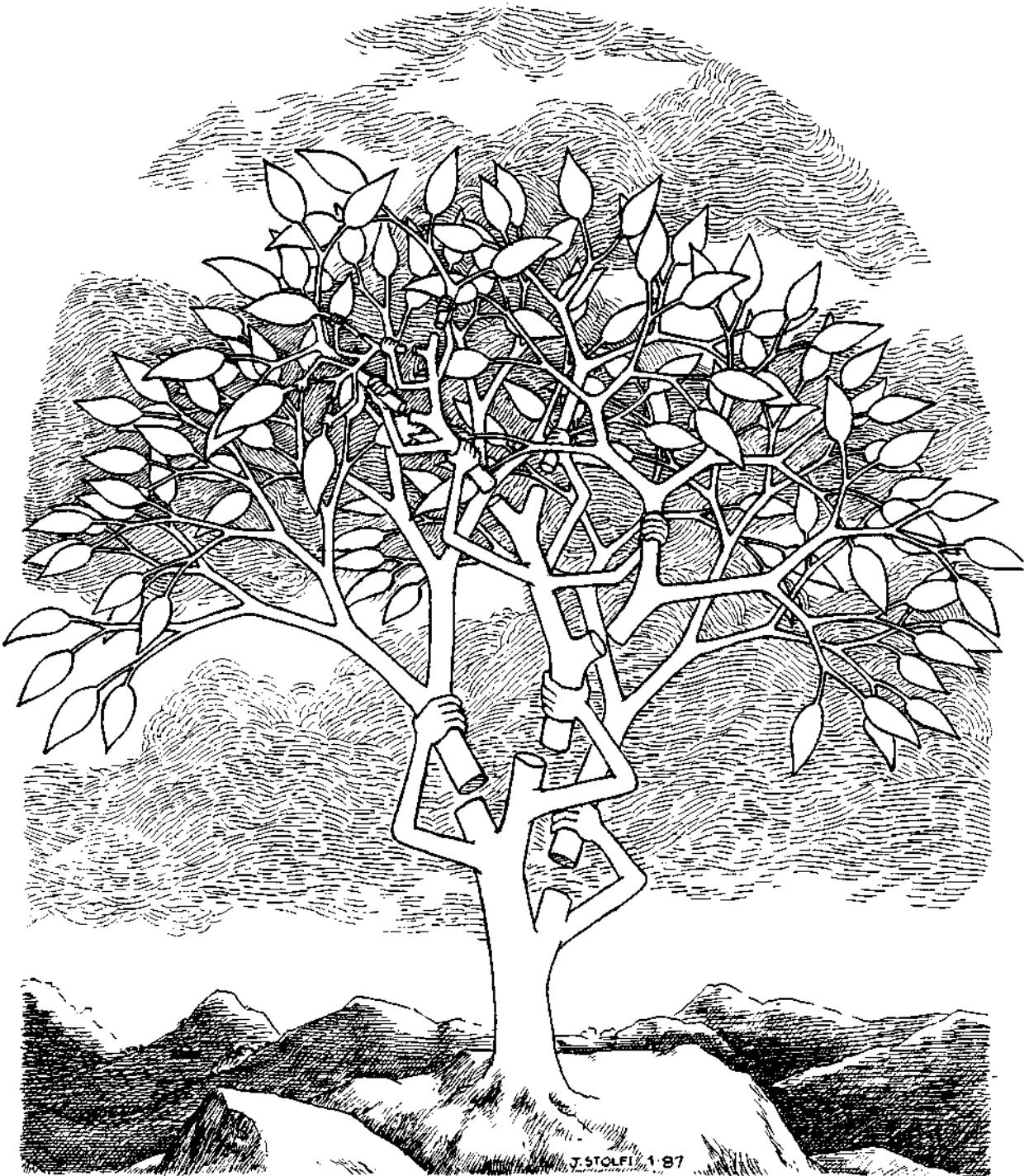
Acknowledgment. I first read of the possibility of such a Trojan horse in an Air Force critique [4] of the security of an early implementation of Multics. I cannot find a more specific reference to this document. I would appreciate it if anyone who can supply this reference would let me know.

REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, apaged time-sharing system for the PDP-10. *Commun. ACM* 15, 3 (Mar. 1972), 135-143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM* 17, (July 1974), 365-375.
4. Unknown Air Force Document.

Author's Present Address: Ken Thompson, AT&T Bell Laboratories, Room 2C-519, 600 Mountain Ave., Murray Hill, NJ 07974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.



A Self-Adjusting Search Tree

ALGORITHM DESIGN

The quest for efficiency in computational methods yields not only fast algorithms, but also insights that lead to elegant, simple, and general problem-solving methods.

ROBERT E. TARJAN

I was surprised and delighted to learn of my selection as corecipient of the 1986 Turing Award. My delight turned to discomfort, however, when I began to think of the responsibility that comes with this great honor: to speak to the computing community on some topic of my own choosing. Many of my friends suggested that I preach a sermon of some sort, but as I am not the preaching kind, I decided just to share with you some thoughts about the work I do and its relevance to the real world of computing.

Most of my research deals with the design and analysis of efficient computer algorithms. The goal in this field is to devise problem-solving methods that are as fast and use as little storage as possible. The efficiency of an algorithm is measured not by programming it and running it on an actual computer, but by performing a mathematical analysis that gives bounds on its potential use of time and space. A theoretical analysis of this kind has one obvious strength: It is independent of the programming of the algorithm, the language in which the program is written, and the specific computer on which the program is run. This means that conclusions derived from such an analysis tend to be of broad applicability. Furthermore, a theoretically efficient algorithm is generally efficient in practice (though of course not always).

But there is a more profound dimension to the design of efficient algorithms. Designing for theoretical efficiency requires a concentration on the important aspects of a problem, so as to avoid redundant

computations and to design data structures that exactly represent the information needed to solve the problem. If this approach is successful, the result is not only an efficient algorithm, but a collection of insights and methods extracted from the design process that can be transferred to other problems. Since the problems considered by theoreticians are generally abstractions of real-world problems, it is these insights and general methods that are of most value to practitioners, since they provide tools that can be used to build solutions to real-world problems.

I shall illustrate algorithm design by relating the historical contexts of two particular algorithms. One is a graph algorithm, for testing the planarity of a graph that I developed with John Hopcroft. The other is a data structure, a self-adjusting form of search tree that I devised with Danny Sleator.

I graduated from CalTech in June 1969 with a B.S. in mathematics, determined to pursue a Ph.D., but undecided about whether it should be in mathematics or computer science. I finally decided in favor of computer science and enrolled as a graduate student at Stanford in the fall. I thought that as a computer scientist I could use my mathematical skills to solve problems of more immediate practical interest than the problems posed in pure mathematics. I hoped to do research in artificial intelligence, since I wished to understand the way reasoning, or at least mathematical reasoning, works. But my course adviser at Stanford was Don Knuth, and I think he had other plans for my future: His first advice to me was to read Volume 1 of his book, *The Art of Computer Programming*.

By June 1970 I had successfully passed my Ph.D.

qualifying examinations, and I began to cast around for a thesis topic. During that month John Hopcroft arrived from Cornell to begin a sabbatical year at Stanford. We began to talk about the possibility of developing efficient algorithms for various problems on graphs.

As a measure of computational efficiency, we settled on the worst-case time, as a function of the input size, of an algorithm running on a sequential random-access machine (an abstraction of a sequential general-purpose digital computer). We chose to ignore constant factors in running time, so that our measure could be independent of any machine model and of the details of any algorithm implementation. An algorithm efficient by this measure tends to be efficient in practice. This measure is also analytically tractable, which meant that we would be able to actually derive interesting results about it.

Other approaches we considered have various weaknesses. In the mid 1960s, Jack Edmonds stressed the distinction between polynomial-time and non-polynomial-time algorithms, and although this distinction led in the early 1970s to the theory of NP-completeness, which now plays a central role in complexity theory, it is too weak to provide much guidance for choosing algorithms in practice. On the other hand, Knuth practiced a style of algorithm analysis in which constant factors and even lower order terms are estimated. Such detailed analysis, however, was very hard to do for the sophisticated algorithms we wanted to study, and sacrifices implementation independence. Another possibility would have been to do average-case instead of worst-case analysis, but for graph problems this is very hard and perhaps unrealistic: Analytically tractable average-case graph models do not seem to capture important properties of the graphs that commonly arise in practice.

Thus, the state of algorithm design in the late 1960s was not very satisfactory. The available analytical tools lay almost entirely unused; the typical content of a paper on a combinatorial algorithm was a description of the algorithm, a computer program, some timings of the program on sample data, and conclusions based on these timings. Since changes in programming details can affect the running time of a computer program by an order of magnitude, such conclusions were not necessarily justified. John and I hoped to help put the design of combinatorial algorithms on a firmer footing by using worst-case running time as a guide in choosing algorithmic methods and data structures.

The focus of our activities became the problem of testing the planarity of a graph. A graph is planar if it can be drawn in the plane so that each vertex

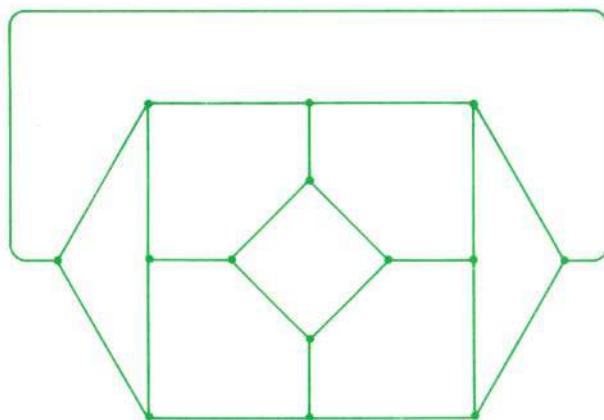


FIGURE 1. A Planar Graph

becomes a point, each edge becomes a simple curve joining the appropriate pair of vertices, and no two edges touch except at a common vertex (see Figure 1).

A beautiful theorem by Casimir Kuratowski states that a graph is planar if and only if it does not contain as a subgraph either the complete graph on five vertices (K_5), or the complete bipartite graph on two sets of three vertices ($K_{3,3}$) (see Figure 2).

Unfortunately, Kuratowski's criterion does not lead in any obvious way to a practical planarity test. The known efficient ways to test planarity involve actually trying to embed the graph in the plane. Either the embedding process succeeds, in which case the graph is planar, or the process fails, in which case the graph is nonplanar. It is not neces-

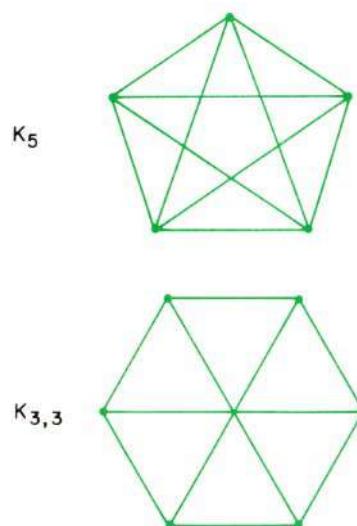


FIGURE 2. The "Forbidden" Subgraphs of Kuratowski

sary to specify the geometry of an embedding; a specification of its topology will do. For example, it is enough to know the clockwise ordering around each vertex of its incident edges.

Louis Auslander and Seymour Parter formulated a planarity algorithm in 1961 called the path addition method. The algorithm is easy to state recursively: Find a simple cycle in the graph, and then remove this cycle to break the rest of the graph into segments. (In a planar embedding, each segment must lie either completely inside or completely outside the embedded cycle; certain pairs of segments are constrained to be on opposite sides of the cycle. See Figure 3.) Test each segment together with the cycle for planarity by applying the algorithm recursively. If each segment passes this planarity test, determine whether the segments can be assigned to the inside and outside of the cycle in a way that satisfies all the pairwise constraints. If so, the graph is planar.

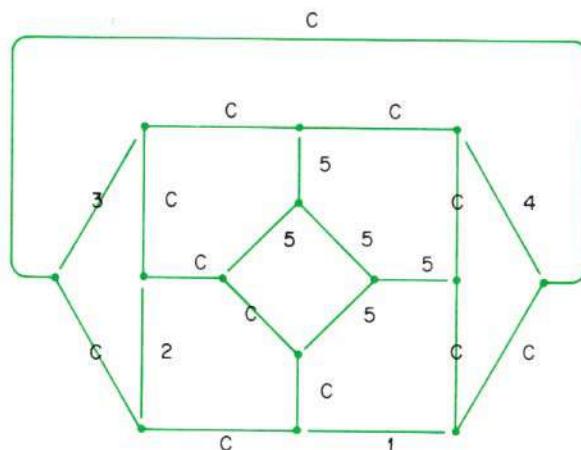
As noted by A. Jay Goldstein in 1963, it is essential to make sure that the algorithm chooses a cycle that produces two or more segments; if only one segment is produced, the algorithm can loop forever. Rob Shirey proposed a version of the algorithm in 1969 that runs in $O(n^3)$ time on an n -vertex graph. John and I worked to develop a faster version of this algorithm.

A useful fact about planar graphs is that they are sparse: Of the $n(n - 1)/2$ edges that an n -vertex graph can contain, a planar graph can contain only at most $3n - 6$ of them (if $n \geq 3$). Thus John and I hoped (audaciously) to devise an $O(n)$ -time planarity test. Eventually, we succeeded.

The first step was to settle on an appropriate graph representation, one that takes advantage of sparsity. We used a very simple representation, consisting of a list, for each vertex, of its incident edges. For a possibly planar graph, such a representation is $O(n)$ in size.

The second step was to discover how to do a necessary bit of preprocessing. The path addition method requires that the graph be biconnected (that is, have no vertices whose removal disconnects the graph). If a graph is not biconnected, it can be divided into maximal biconnected subgraphs, or biconnected components. A graph is planar if and only if all of its biconnected components are planar. Thus planarity can be tested by first dividing a graph into its biconnected components and then testing each component for planarity. We devised an algorithm for finding the biconnected components of an n -vertex, m -edge graph in $O(n + m)$ time. This algorithm uses depth-first search, a systematic way of exploring a graph and visiting each vertex and edge.

We were now confronted with the planarity test



Segments 1 and 2 must be embedded on opposite sides of C , as must segments 1 and 3, 1 and 4, 2 and 5, 3 and 5, and 4 and 5.

FIGURE 3. Division of the Graph in Figure 1 by Removal of Cycle C

itself. There were two main problems to be solved. If the path addition method is formulated iteratively instead of recursively, the method becomes one of embedding an initial cycle and then adding one path at a time to a growing planar embedding. Each path is disjoint from earlier paths except at its end vertices. There is in general more than one way to embed a path, and embedding of later paths may force changes in the embedding of earlier paths. We needed a way to divide the graph into paths and a way to keep track of the possible embeddings of paths so far processed.

After carefully studying the properties of depth-first search, we developed a way to generate paths in $O(n)$ time using depth-first search. Our initial method for keeping track of alternative embeddings used a complicated and ad hoc nested structure. For this method to work correctly, paths had to be generated in a specific, dynamically determined order. Fortunately, our path generation algorithm was flexible enough to meet this requirement, and we obtained a planarity algorithm that runs in $O(n \log n)$ time.

Our attempts to reduce the running time of this algorithm to $O(n)$ failed, and we turned to a slightly different approach, in which the first step is to generate all the paths, the second step is to construct a graph representing their pairwise embedding constraints, and the third step is to color this constraint graph with two colors, corresponding to the two possible ways to embed each path. The problem with

this approach is that there can be a quadratic number of pairwise constraints. Obtaining a linear time bound requires computing explicitly only enough constraints so that their satisfaction guarantees that all remaining constraints are satisfied as well. Working out this idea led to an $O(n)$ -time planarity algorithm, which became the subject of my Ph.D. dissertation. This algorithm is not only theoretically efficient, but fast in practice: My implementation, written in Algol W and run on an IBM 360/67, tested graphs with 900 vertices and 2694 edges in about 12 seconds. This was about 80 times faster than any other claimed time bound I could find in the literature. The program is about 500 lines long, not counting comments.

This is not the end of the story, however. In preparing a description of the algorithm for journal publication, we discovered a way to avoid having to construct and color a graph to represent pairwise embedding constraints. We devised a data structure, called a pile of twin stacks, that can represent all possible embeddings of paths so far processed and is easy to update to reflect the addition of new paths. This led to a simpler algorithm, still with an $O(n)$ time bound. Don Woods, a student of mine, programmed the simpler algorithm, which tested graphs with 7000 vertices in 8 seconds on an IBM 370/168. The length of the program was about 250 lines of Algol W, of which planarity testing required only about 170, the rest being used to actually construct a planar representation.

From this research we obtained not only a fast planarity algorithm, but also an algorithmic technique (depth-first search) and a data structure (the pile of twin stacks) useful in solving many other problems. Depth-first search has been used in efficient algorithms for a variety of graph problems; the pile of twin stacks has been used to solve problems in sorting and in recognizing codes for planar self-intersecting curves.

Our algorithm is not the only way to test planarity in $O(n)$ time. Another approach, by Abraham Lempel, Shimon Even, and Israel Cederbaum, is to build an embedding by adding one vertex and its incident edges at a time. A straightforward implementation of this algorithm gives an $O(n^2)$ time bound. When John and I were doing our research, we saw no way to improve this bound, but later work by Even and myself and by Kelly Booth and George Lueker yielded an $O(n)$ -time version of this algorithm. Again, the method combines depth-first search, in a preprocessing step to determine the order of vertex addition, with a complicated data structure, the PQ-tree of Booth and Lueker, for keeping track of

possible embeddings. (This data structure itself has several other applications.)

There is yet a third $O(n)$ -time planarity algorithm. I only recently discovered that a novel form of search tree called a finger search tree, invented in 1977 by Leo Guibas, Mike Plass, Ed McCreight, and Janet Roberts, can be used in the original planarity algorithm that John and I developed to improve its running time from $O(n \log n)$ to $O(n)$. Deriving the time bound requires the solution of a divide-and-conquer recurrence. Finger search trees had no particularly compelling applications for several years after they were invented, but now they do, in sorting and computational geometry, as well as in graph algorithms.

Ultimately, choosing the correct data structures is crucial to the design of efficient algorithms. These structures can be complicated, and it can take years to distill a complicated data structure or algorithm down to its essentials. But a good idea has a way of eventually becoming simpler and of providing solutions to problems other than those for which it was intended. Just as in mathematics, there are rich and deep connections among the apparently diverse parts of computer science.

I want to shift the emphasis now from graph algorithms to data structures by reflecting a little on whether worst-case running time is an appropriate measure of a data structure's efficiency. A graph algorithm is generally run on a single graph at a time. An operation on a data structure, however, does not occur in isolation; it is one in a long sequence of similar operations on the structure. What is important in most applications is the time taken by the entire sequence of operations rather than by any single operation. (Exceptions occur in real-time applications where it is important to minimize the time of each individual operation.)

We can estimate the total time of a sequence of operations by multiplying the worst-case time of an operation by the number of operations. But this may produce a bound that is so pessimistic as to be useless. If in this estimate we replace the worst-case time of an operation by the average-case time of an operation, we may obtain a tighter bound, but one that is dependent on the accuracy of the probability model. The trouble with both estimates is that they do not capture the correlated effects that successive operations can have on the data structure. A simple example is found in the pile of twin stacks used in planarity testing: A single operation among a sequence of n operations can take time proportional to n , but the total time for n operations in a sequence is always $O(n)$. The appropriate measure in such a situ-

ation is the amortized time, defined to be the time of an operation averaged over a worst-case sequence of operations.¹ In the case of a pile of twin stacks, the amortized time per operation is $O(1)$.

A more complicated example of amortized efficiency is found in a data structure for representing disjoint sets under the operation of set union. In this case, the worst-case time per operation is $O(\log n)$, where n is the total size of all the sets, but the amortized time per operation is for all practical purposes constant but in theory grows very slowly with n (the amortized time is a functional inverse of Ackermann's function).

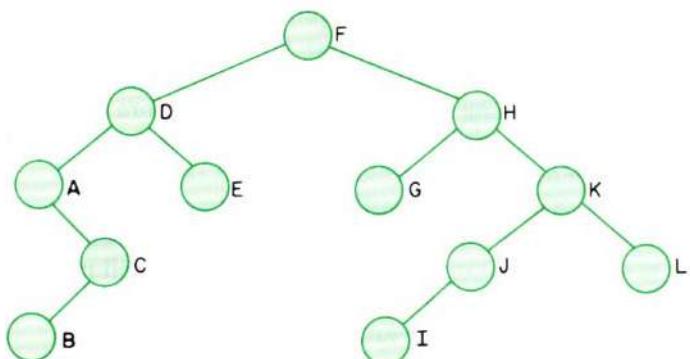
These two examples illustrate the use of amortization to provide a tighter analysis of a known data structure. But amortization has a more profound use. In designing a data structure to reduce amortized running time, we are led to a different approach than if we were trying to reduce worst-case running time. Instead of carefully crafting a structure with exactly the right properties to guarantee a good worst-case time bound, we can try to design simple, local restructuring operations that improve the state of the structure if they are applied repeatedly. This approach can produce "self-adjusting" or "self-organizing" data structures that adapt to fit their usage and have an amortized efficiency close to optimum among a broad class of competing structures.

The splay tree, a self-adjusting form of binary search tree that Danny Sleator and I developed, is one such data structure. In order to discuss it, I need first to review some concepts and results about search trees. A binary tree is either empty or consists of a node called the root and two node-disjoint binary trees, called the left and right subtrees of the root. The root of the left (right) subtree is the left (right) child of the root of the tree. A binary search tree is a binary tree containing the items from a totally ordered set in its nodes, one item per node, with the items arranged in symmetric order; that is, all items in the left subtree are less than the item in the root, and all items in the right subtree are greater (see Figure 4).

A binary search tree supports fast retrieval of the items in the set it represents. The retrieval algorithm, based on binary search, is easy to state recursively. If the tree is empty, stop: The desired item is not present. Otherwise, if the root contains the desired item, stop: The item has been found. Otherwise, if the desired item is greater than the one in the root, search the left subtree; if the desired item is less than the one in the root, search the right

subtree. Such a search takes time proportional to the depth of the desired item, defined to be the number of nodes examined during the search. The worst-case search time is proportional to the depth of the tree, defined to be the maximum node depth.

Other efficient operations on binary search trees include inserting a new item and deleting an old one (the tree grows or shrinks by a node, respectively). Such trees provide a way of representing static or dynamically changing sorted sets. Although in many situations a hash table is the preferred representation, since it supports retrieval in $O(1)$ time on the average, a search tree is the data structure of choice if the ordering among items is important. For example, in a search tree it is possible in a single search to find the largest item smaller than a given one, something that in a hash table requires examining all the items. Search trees also support even more complicated operations, such as retrieval of all items between a given pair of items (range retrieval) and concatenation and splitting of lists.

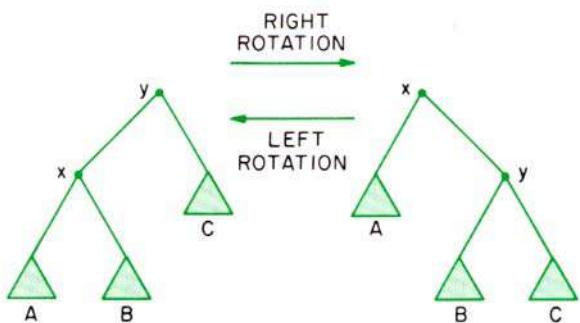


The items are ordered alphabetically.

FIGURE 4. A Binary Search Tree

An n -node binary tree has some node of depth at least $\log_2 n$; thus, for any binary search tree the worst-case retrieval time is at least a constant times $\log n$. One can guarantee an $O(\log n)$ worst-case retrieval time by using a balanced tree. Starting with the discovery of height-balanced trees by Georgii Adelson-Velskii and Evgenii Landis in 1962, many kinds of balanced trees have been discovered, all based on the same general idea. The tree is required to satisfy a balance constraint, some local property that guarantees a depth of $O(\log n)$. The balance constraint is restored after an update, such as an insertion or deletion, by performance of a sequence of local transformations on the tree. In the case of binary trees, each transformation is a rotation, which takes constant time, changes the depths of certain

¹See R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Disc. Math.* 6, 2 (Apr. 1985), 306–318, for a thorough discussion of this concept.



The right rotation is at node x , and the inverse left rotation at node y . The triangles denote arbitrary subtrees, possibly empty. The tree shown can be part of a larger tree.

FIGURE 5. A Rotation in a Binary Search Tree

nodes, and preserves the symmetric order of the items (see Figure 5).

Although of great theoretical interest, balanced search trees have drawbacks in practice. Maintaining the balance constraint requires extra space in the nodes for storage of balance information and requires complicated update algorithms with many cases. If the items in the tree are accessed more or less uniformly, it is simpler and more efficient in practice not to balance the tree at all; a random sequence of insertions will produce a tree of depth $O(\log n)$. On the other hand, if the access distribution is significantly skewed, then a balanced tree will not minimize the total access time, even to within a constant factor. To my knowledge, the only kind of balanced tree extensively used in practice is the *B-tree* of Rudolph Bayer and Ed McCreight, which generally has many more than two children per node and is suitable for storing very large sets of data on secondary storage media such as disks. *B-trees* are useful because the benefits of balancing increase with the number of children per node and with the corresponding decrease in maximum depth. In practice, the maximum depth of a *B-tree* is a small constant (three or four).

The invention of splay trees arose out of work I did with two of my students, Sam Bent and Danny Sleator, at Stanford in the late 1970s. I had returned to Stanford as a faculty member after spending some time at Cornell and Berkeley. Danny and I were attempting to devise an efficient algorithm to compute maximum flows in networks. We reduced this problem to one of constructing a special kind of search tree, in which each item has a positive weight and the time it takes to retrieve an item depends on its weight, heavier items being easier to access than lighter ones. The hardest requirement to meet was the capacity for performing drastic update

operations fast; each tree was to represent a list, and we needed to allow for list splitting and concatenation. Sam, Danny, and I, after much work, succeeded in devising an appropriate generalization of balanced search trees, called biased search trees, which became the subject of Sam's Ph.D. thesis. Danny and I combined biased search trees with other ideas to obtain the efficient maximum flow algorithm we had been seeking. This algorithm in turn was the subject of Danny's Ph.D. thesis. The data structure that forms the heart of this algorithm, called a dynamic tree, has a variety of other applications.

The trouble with biased search trees and with our original version of dynamic trees is that they are complicated. One reason for this is that we had tried to design these structures to minimize the worst-case time per operation. In this we were not entirely successful; update operations on these structures only have the desired efficiency in the amortized sense. After Danny and I both moved to AT&T Bell Laboratories at the end of 1980, he suggested the possibility of simplifying dynamic trees by explicitly seeking only amortized efficiency instead of worst-case efficiency. This idea led naturally to the problem of designing a "self-adjusting" search tree that would be as efficient as balanced trees but only in the amortized sense. Having formulated this problem, it took us only a few weeks to come up with a candidate data structure, although analyzing it took us much longer (the analysis is still incomplete).

In a splay tree, each retrieval of an item is followed by a restructuring of the tree, called a splay operation. (Splay, as a verb, means "to spread out.") A splay moves the retrieved node to the root of the tree, approximately halves the depth of all nodes accessed during the retrieval, and increases the depth of any node in the tree by at most two. Thus, a splay makes all nodes accessed during the retrieval much easier to access later, while making other nodes in the tree at worst a little harder to access.

The details of a splay operation are as follows (see Figure 6): To splay at a node x , repeat the following step until node x is the tree root.

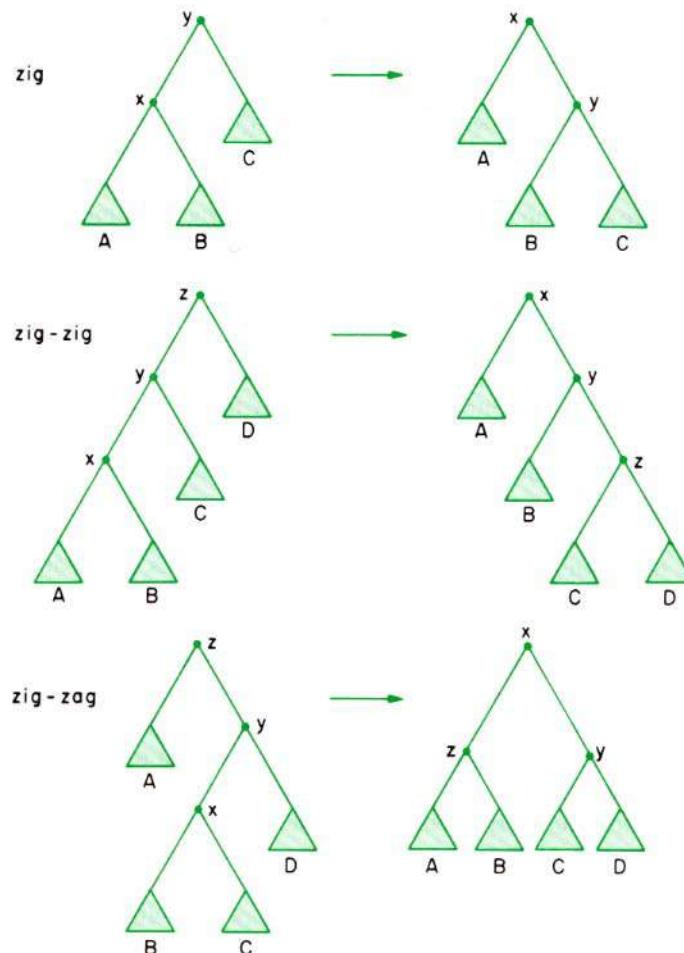
Splay Step. Of the following three cases, apply the appropriate one:

Zig case. If x is a child of the tree root, rotate at x (this case is terminal).

Zig-zig case. If x is a left child and its parent is a left child, or if both are right children, rotate at the parent of x and then at x .

Zig-zag case. If x is a left child and its parent is a right child, or vice versa, rotate at x and then again at x .

As Figure 7 suggests, a sequence of costly retrievals in an originally very unbalanced splay tree will



In the zig-zig and zig-zag cases, the tree shown can be part of a larger tree.

FIGURE 6. The Cases of a Splay Step

quickly drive it into a reasonably balanced state. Indeed, the amortized retrieval time in an n -node splay tree is $O(\log n)$. Splay trees have even more striking theoretical properties. For an arbitrary but sufficiently long sequence of retrievals, a splay tree is as efficient to within a constant factor as an optimum static binary search tree expressly constructed to minimize the total retrieval time for the given sequence. Splay trees perform as well in an am-

ortized sense as biased search trees, which allowed Danny and I to obtain a simplified version of dynamic trees, our original goal.

On the basis of these results, Danny and I conjecture that splay trees are a universally optimum form of binary search tree in the following sense: Consider a fixed set of n items and an arbitrary sequence of $m \geq n$ retrievals of these items. Consider any algorithm that performs these retrievals by starting

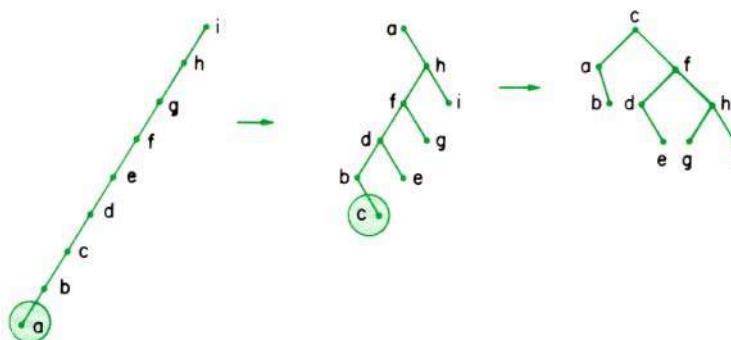


FIGURE 7. A Sequence of Two Costly Splays on an Initially Unbalanced Tree

with an initial binary search tree, performing each retrieval by searching from the root in the standard way, and intermittently restructuring the tree by performing rotations anywhere in it. The cost of the algorithm is the sum of the depths of the retrieved items (when they are retrieved) plus the total number of rotations. We conjecture that the total cost of the splaying algorithm, starting with an arbitrarily bad initial tree, is within a constant factor of the minimum cost of any algorithm. Perhaps this conjecture is too good to be true. One additional piece of evidence supporting the conjecture is that accessing all the items of a binary search tree in sequential order using splaying takes only linear time.

In addition to their intriguing theoretical properties, splay trees have potential value in practice. Splaying is easy to implement, and it makes tree update operations such as insertion and deletion simple as well. Preliminary experiments by Doug Jones suggest that splay trees are competitive with all other known data structures for the purpose of implementing the event list in a discrete simulation system. They may be useful in a variety of other applications, although verifying this will require much systematic and careful experimentation.

The development of splay trees suggests several conclusions. Continued work on an already-solved problem can lead to major simplifications and additional insights. Designing for amortized efficiency can lead to simple, adaptive data structures that are more efficient in practice than their worst-case-efficient cousins. More generally, the efficiency measure chosen suggests the approach to be taken in tackling an algorithmic problem and guides the development of a solution.

I want to make a few comments about what I think the field of algorithm design needs. It is trite to say that we could use a closer coupling between theory and practice, but it is nevertheless true. The world of practice provides a rich and diverse source of problems for researchers to study. Researchers can provide practitioners with not only practical algorithms for specific problems, but broad approaches and general techniques that can be useful in a variety of applications.

Two things would support better interaction between theory and practice. One is much more work on experimental analysis of algorithms. Theoretical analysis of algorithms rests on sound foundations. This is not true of experimental analysis. We need a disciplined, systematic, scientific approach. Experimental analysis is in a way much harder than theoretical analysis because experimental analysis requires the writing of actual programs, and it is hard to avoid introducing bias through the coding process or through the choice of sample data.

Another need is for a programming language or notation that will simultaneously be easy to understand by a human and efficient to execute on a machine. Conventional programming languages force the specification of too much irrelevant detail, whereas newer very-high-level languages pose a challenging implementation task that requires much more work on data structures, algorithmic methods, and their selection.

There is now tremendous ferment within both the theoretical and practical communities over the issue of parallelism. To the theoretician, parallelism offers a new model of computation and thereby changes the ground rules of theoretical analysis. To the practitioner, parallelism offers possibilities for obtaining tremendous speedups in the solution of certain kinds of problems. Parallel hardware will not make theoretical analysis unimportant, however. Indeed, as the size of solvable problems increases, asymptotic analysis becomes more, not less, important. New algorithms will be developed that exploit parallelism, but many of the ideas developed for sequential computation will transfer to the parallel setting as well. It is important to realize that not all problems are amenable to parallel solution. Understanding the impact of parallelism is a central goal of much of the research in algorithm design today.

I do research in algorithm design not only because it offers possibilities for affecting the real world of computing, but because I love the work itself, the rich and surprising connections among problems and solutions it reveals, and the opportunity it provides to share with creative, stimulating, and thoughtful colleagues in the discovery of new ideas. I thank the Association for Computing Machinery and the entire computing community for this award, which recognizes not only my own ideas, but those of the individuals with whom I have had the pleasure of working. They are too many to name here, but I want to acknowledge all of them, and especially John Hopcroft and Danny Sleator, for sharing their ideas and efforts in this process of discovery.

CR Categories and Subject Descriptors: A.0 [General Literature]: General—biographies/autobiographies; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems; G.2.2 [Discrete Mathematics]: Graph Theory; K.2 [Computing Milieux]: History of Computing—people

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: John E. Hopcroft, Robert E. Tarjan, Turing Award

Author's Present Addresses: Robert E. Tarjan, Computer Science Dept., Princeton University, Princeton, NJ 08544; and AT&T Bell Laboratories, Murray Hill, NJ 07974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

(Handwriting)

To Fernando J. Corbato
for his work in organizing the
concepts and leading the development
of the general-purpose large-scale
time-sharing and resource-sharing
computer systems CTSS and MULTICS

On

Building Systems That Will Fail

FERNANDO J. CORBATO

It is an honor and a pleasure to accept the Alan Turing Award. My own work has been on computer systems, and that will be my theme. The essence of systems is that they are integrating efforts, requiring broad knowledge of the problem area to be addressed, and the detailed knowledge required is rarely held by one person. Thus the work of systems is usually done by teams. Hence I am accepting this award on behalf of the many with whom I have worked as much as for myself. It is not practical to name all the individuals who contributed. Nevertheless, I would like to give special mention to Marjorie Daggert and Bob Daley for their parts in the birth of CTSS and to Bob Fano and the late Ted Glaser for their critical contributions to the development of the Multics System.

Let me turn now to the title of this talk: "On Building Systems That Will Fail." Of course the title I chose was a teaser. I considered and discarded some alternate titles: "On Building Messy Systems," but it seemed too frivolous and suggests there is no systematic approach. "On Mastering System Complexity" sounded like I have all the answers. The title that came closest, "On Building Systems that are likely to have Failures" did not have the nuance of inevitability that I wanted to suggest.

What I am really trying to address is the class of systems that for want of a better phrase, I will call "ambitious systems." It almost goes without saying that ambitious systems never quite work as expected. Things usually go wrong—sometimes in dramatic ways. And this leads me to my main thesis, namely, that the question to ask when designing such systems is not: "if something will go wrong, but when it will go wrong?"

Some Examples

Now, ambitious systems that fail are really much more common than we may realize. In fact in some circumstances we strive for them, revelling

in the excitement of the unexpected. For example, let me remind you of our national sport of football. The whole object of the game is for each team to play at the limit of its abilities. Besides the sheer physical skill required, one has the strategic intricacies, the ability to audibilize, and the quickness to react to the unexpected—all a deep part of the game. Of course, occasionally one team approaches perfection, all the plays work, and the game becomes dull.

Another example of a system that is too ambitious for perfection is military warfare. The same elements are there with opposing sides having to constantly improvise and deal with the unexpected. In fact we get from the military that wonderful acronym, SNAFU, which is politely translated as "situation normal, all fouled up." And if any of you are still doubtful, consider how rapidly the phrases "precision bombing" and "surgical strikes" are replaced by "the fog of war" and "casualties from friendly fire" as soon as hostilities begin.

On a somewhat more whimsical note, let me offer driving in Boston as an example of systems that will fail. Automobile traffic is an excellent case of distributed control with a common set of protocols called traffic regulations. The Boston area is notorious for the free interpretations drivers make of these pesky regulations, and perhaps the epitome of it occurs in the arena of the traffic rotary. A case can be made for rotaries. They are efficient. There is no need to wait for sluggish traffic signals. They are direct. And they offer great opportunities for creative improvisation, thereby adding zest to the sport of driving.

One of the most effective strategies is for a driver approaching a rotary to rigidly fix his or her head, staring forward, of course, secretly using peripheral vision to the limit. It is even more effective if the driver on entering the rotary, speeds up, and some drivers embellish this last step by adopting a look of maniacal glee. The effect is, of

course, one of intimidation, and a pecking order quickly develops.

The only reason there are not more accidents is that most drivers have a second component to the strategy, namely, they assume everyone else may be crazy—they are often correct—and every driver is really prepared to stop with inches to spare. Again we see an example of a system where ambitious tactics and prudent caution lead to an effective solution.

So far, the examples I have given may suggest that failures of ambitious systems come from the human element and that at least the technical parts of the system can be built correctly. In particular, turning to computer systems, it is only a matter of getting the code debugged. Some assume rigorous testing will do the job. Some put their hopes in proving program correctness. But unfortunately, there are many cases for which none of these techniques will always work [1]. Let me offer a modest example illustrated in Figure 1.

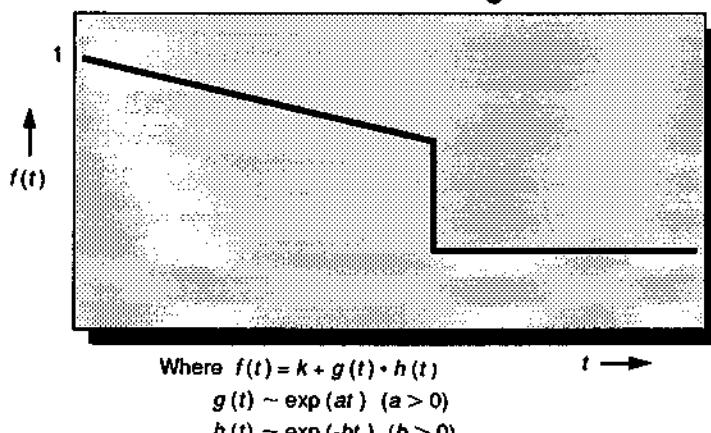
Consider the case of an elaborate numerical calculation with a variable, f , representing some physical value, being calculated for a set of points over a range of a parameter, t . Now the property of physical variables is that they normally do not exhibit abrupt changes or discontinuities.

So what has happened here? If we look at the expression for f , we see it is the result of a constant, k , added to the product of two other functions, g and h . Looking further, we see that the function g has a behavior that is exponentially increasing with t . The function h , on the other hand, is exponentially decreasing with t . The resultant product of g and h is almost constant with increasing t until an abrupt jump occurs and the curve for f goes flat.

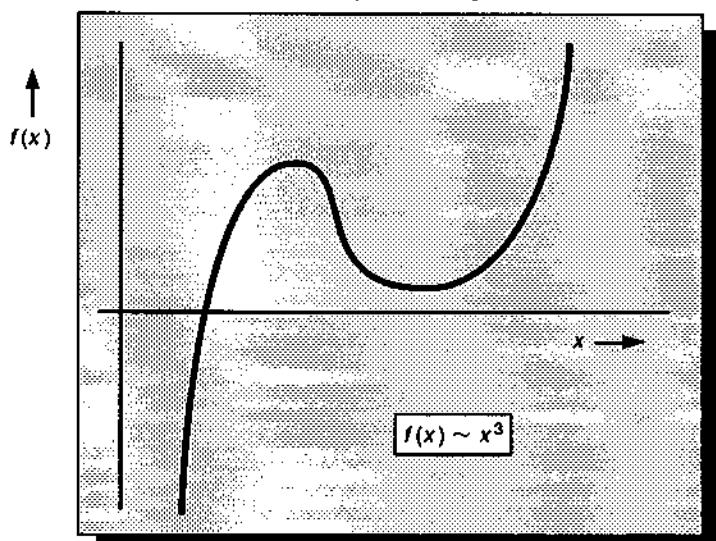
What has gone wrong? The answer is that there has been floating-point underflow at the critical point in the curve, i.e., the representation of the negative exponent has exceeded the field size in the floating-

(I'm thinking)

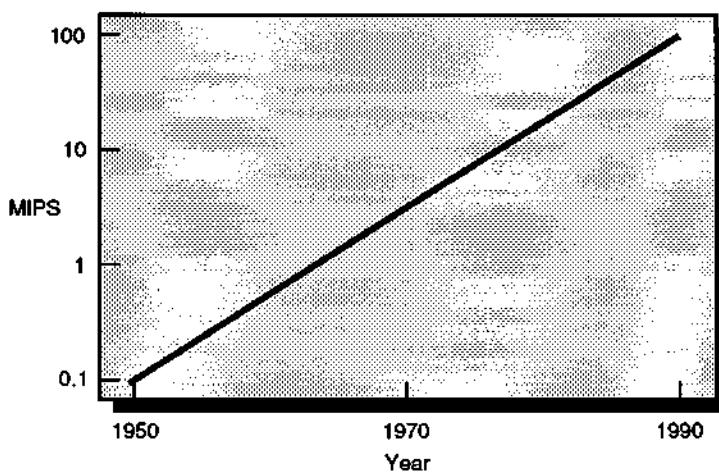
A Subtle Bug



... Why Mishaps?



Performance



point representation for this particular computer, and the hardware has automatically set the value for the function h to zero. Often this is reasonable since small numbers are correctly approximated by zero—but not in this case, where our results are grossly wrong. Worse yet, since the computation of f might be internal, it is easy to imagine that the failure shown here would not be noticed.

Because correctly handling the pathology that this example represents is an extra engineering bother, it should not be surprising that the problem of underflow is frequently ignored. But the larger lesson to be learned from this example is that subtle mistakes are very difficult to avoid and to some extent are inevitable.

I encountered my next example when I was a graduate student programming on the pioneering Whirlwind computer. One night while awaiting my turn to use it, the graduate student before me began complaining of how "tough" some of his calculations were. He said he was computing the vibrational frequencies of a particular wing structure for a series of cases. In fact, his equations were cubics, and he was using the iterative Newton-Raphson method. For reasons he did not understand, his method was finding one of the roots, but not "converging" for the others. He was trying to fix this situation by changing his program so that when he encountered one of these tough roots, the program would abandon the iteration after a fixed number of tries.

Now there were several things wrong: First, the coefficients to his cubic equations were based on ex-

FIGURE 1
Debugging the Code

FIGURE 2
Nonconverging Iterative Method
Caused by Poor Root Value

FIGURE 3
Performance of a Top-of-the-Line
Computer by Decade

perimental data and some of his points were simply bad. Therefore, as Figure 2 illustrates, he only had one real root and a pair of imaginaries. Thus his iterative method could never converge for the second and third roots and the value of his first root was pure garbage. Second, cubic equations have an exact analytic closed form solution so that it was entirely unnecessary to use an iterative method. And third, based on his incomplete model and understanding of what was happening, he exercised very poor judgment in patching his program to ignore values that were seemingly difficult to compute.

Ambitious System Properties

Let me turn next to some of the general properties of ambitious systems. First, they are often vast and have significant organizational structures going beyond that of simple replication. Second, they are frequently complicated or elaborate and are too much for even a small group to develop. Third, if they really are ambitious, they are pushing the envelope of what people know how to do, and as a result there is always a level of uncertainty about when completion is possible. Because one has to be an optimist to begin an ambitious project, it is not surprising that underestimation of completion time is the norm. Fourth, ambitious systems when they work, often break new ground, offer new services and soon become indispensable. Finally, it is often the case that ambitious systems by virtue of having opened up a new domain of usage, invite a flood of improvements and changes.

Now one could argue that ambitious systems are really only difficult the first time or two. It is really only a matter of learning how to do it. Once one has, then one simply draws up the appropriate PERT charts, hires good managers, ensures an adequate budget and gets on with it. Perhaps there are some instances where this works, but at least in the area of computer sys-

tems, there is a fundamental reason it does not.

A key reason we cannot seem to get ambitious systems right is change. The computer field is intoxicated with change. We have seen galloping growth over a period of four decades and it still does not seem to be slowing down. The field is not mature yet and already it accounts for a significant percentage of the Gross National Product both directly and indirectly. More importantly the computer revolution—this second industrial revolution—has changed our life-styles and allowed the growth of countless new application areas. And all this change and growth not only has changed the world we live in, but has raised our expectations, spurring on increasingly ambitious systems in such diverse areas as airline reservations, banking, credit cards, and air traffic control to name only a few.

Behind the incredible growth of the computer industry is, of course, the equally mind-boggling change that has occurred in the raw performance of digital logic. Figure 3, which is not precise and which many of you have seen before in some form, gives the performance of a top-of-the-line computer by decade. The ordinate in MIPS is logarithmic as you can see. In particular in the last decade, the graph becomes problem dependent so that the upper right-hand end of the line should break up into some sort of whiskers as more and more computers are tailored for special applications and for parallelism.

Complicating matters too is that parallelism is not a solution for every problem. Certain calculations that are intrinsically serial, such as rocket trajectories, derive very limited benefit from parallel computers. And one of course is reminded of the old joke about the Army way of speeding up pregnancy by having nine women spend one month at the task.

As Figure 4 makes clear, it is not just performance that has fueled growth but rather cost/perform-

mance, or simply put, favorable economics. The graph is an oversimplification, but represents the cost for a given performance computer model over the last four decades. Again the ordinate is logarithmic, going from 10 million dollars in 1950 down to one thousand dollars in 1990. As we approach the present, corresponding to a personal computer, the graph really should become more complicated since one consequence of computers becoming super-cheap is that increasingly, they are being embedded in other equipment. The modern automobile is but one example. And it

**remains
to be seen
how general-
purpose the
current wave
of palm-sized
computers
will be with
their stylus
inputs .**

Further, when we look at a photograph taken around 1960 of a "machine room" staffed with one lone operator, we are reminded of the fantastic changes that have occurred in computer technology. The boxes are huge, shower-stall-sized, and the overall impression is of some small factory. You were supposed to be impressed and the operator was expected to maintain decorum by wearing a necktie. And if he did not, at least you could be sure an IBM maintenance engineer would.

Another reminder of the immense technological change which has occurred is in the physical dimensions of the main memories of computers. For example, if one looks at old photographs taken in the mid-1950s of core memory sys-

[Handwritten]

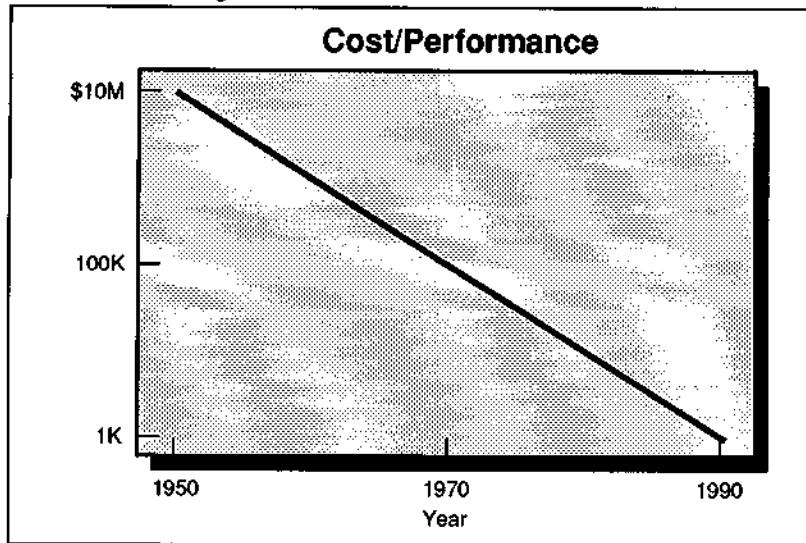


FIGURE 4
Cost for Given-Performance Computer Model over Four Decades

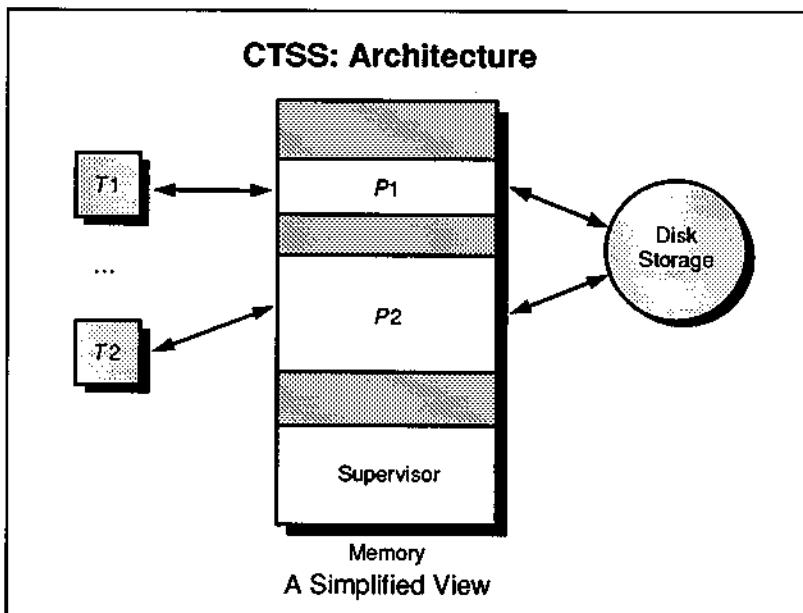


FIGURE 5
Input/Output of User Programs

tems, one typically sees a core memory plane roughly the size of a tennis racquet head which could hold about 1,000 bits of information. Contrast that with today's 4megabit memory chips that are smaller than one's thumb.

The basis of the Award today is largely for my work on two pioneering time-sharing systems,

CTSS [5, 6] and Multics [7, 9]. Indeed, it is from my involvement with those two systems that I gained the system-building perspective I am offering. It therefore seems appropriate to take a brief retrospective look at these two systems as examples of ambitious systems and to explore the reasons why the complexity of the tasks involved made it almost impossible to build the systems correctly the first time [2].

CTSS, The Compatible Time-Sharing System

Looking first at CTSS, let us remember the dark ages that existed then. This was the early 1960s. The computers of the day were big and expensive, and the administrators of computing centers felt obliged to husband the precious resource. Users, i.e., programmers, were expected to submit a computing job as a deck of punched cards. These were then combined into a batch with other jobs onto a magnetic tape and the tape was processed by the computer operating system. It had all the glamour and excitement of dropping one's clothes off at a laundromat.

The problem was that even for a trivial input typing mistake, the job would be aborted. Time-sharing, as most of you know, was the solution to the problem of not being able to interact with computers. The general vision of modern time-sharing was primarily spelled out by John McCarthy, who I am pleased to note is a featured speaker at this conference. In England, Christopher Strachey independently came up with a limited kind of interactive computing, but it was aimed mostly at debugging. Soon there were many groups around the country developing various forms of interactive computing, but in almost all cases, the resulting systems had significant limitations.

It was in this context that my own group developed our version of the time-sharing vision. We called it The Compatible Time-Sharing System, or CTSS for short. Our initial aspirations were modest. First, the system was meant to be a demonstration prototype before more ambitious designs being attempted by others could be implemented. Second, it was intended to handle general-purpose programming. And third, it was meant to make it possible to run most of the large body of software that had been developed over the years in the batch-processing environment. Hence the name.

The basic scheme used to run

CTSS was simple. The supervisor program, which was always in main memory, would commutate among the user programs, running each in turn for a brief interval with the help of an interval timer. As Figure 5 indicates, user programs could do input/output with the typewriter-like terminals and with the disk storage unit as well.

But the diagram is oversimplified. The key difficulty was that main memory was in short supply and not all the programs of the active users could remain in memory at once. Thus the supervisor program not only had to move programs to and from the disk storage unit, but it also had to act as an intermediary for all I/O initiated by user programs. Thus all the I/O lines should only point to the supervisor program.

As a further complication, the supervisor program had to prevent user programs from trampling over one another. To do this required special hardware modifications to the processor such that there were memory bound registers that could only be set by the supervisor. Nevertheless, despite all the complications, the simplicity of the initial supervisor program allowed it to occupy about 22 Kbytes of storageless storage than required for the text of this talk!

Most of the battles of creating CTSS involved solving problems which at the time did not have standard solutions. For example: There were no standard terminals. There were no simple modems. I/O to the computer was by word and not by character, and worse yet, did not accommodate lower case letters. The computers of the day had neither interrupt timers nor calendar clocks. There was no way to prevent user programs from issuing raw I/O instructions at random. There was no memory protection scheme. And, there was no easy way to store large amounts of data with relatively rapid random access.

The overall result of building CTSS was to change the style of computing, but there were several

effects that seem worth noting. One of the most important was that we discovered that writing interactive software was quite different from software for batch operation and even today, in this era of personal computers, the evolution of interactive interfaces continues.

In retrospect, several design decisions contributed to the success of CTSS, but two were key. First, we could do general-purpose programming and, in particular, develop new supervisor software using the system itself. Second, by making the system able to accommodate older batch code, we inherited a wealth of older software ready-to-go.

One important consequence of developing CTSS was that for the first time, users had persistent on-line storage of programs and data. Suddenly the issues of privacy, protection and backup of information had to be faced. Another byproduct of the development was that because we operated terminals via modems, remote operation became the norm. Also, the new-found freedom of keeping information on-line in the central file system suddenly made it especially convenient for users to share and exchange information among themselves.

And there were surprises too. To our dismay, users who had been enduring several-hour waits between jobs run under batch processing were suddenly restless when response times were more than a second. Moreover, many of the simplifying assumptions that had allowed CTSS to be built so simply, such as a one-level file system, suddenly began to chafe. It seemed like the more we did, the more users wanted.

There are two other observations that can be made about the CTSS system. First, it lasted far longer than we expected. Although CTSS had been demonstrated in primitive form in November 1961, it was not until 1963 that it came into wide use as the vehicle of a Project MAC Summer Study. For a time there

were two copies of the system hardware, but by 1973 the last copy was turned off and scrapped primarily because the maintenance costs of the IBM 7094 hardware had become prohibitively expensive, and up to the bitter end, there were users desperately trying to get in a few last hours of use.

Second, the then-new transistors and large random-access disk files were absolutely critical to the success of time-sharing. The previous generation of vacuum tubes was simply too unreliable for sustained real-time operation and, of course, large disk files were crucial for the central storage of user programs and data.

A Mishap

My central theme is to try to convince you that

**when
you have a
multitude of
novel issues
to contend
with while
building a
system,
mistakes are
inevitable.**

And indeed, we had a beauty while using CTSS. Let me describe it:

What happened was that one afternoon at Project MAC, where CTSS was being used as the main time-sharing workhorse, any user who logged in, found that instead of the usual message-of-the-day typing out on his or her terminal, he had the entire file of user passwords. This went on for 15 or 20 minutes, until one particularly conscientious user called the system administrator and began the conversation with "Did you know that . . . ?" Needless to say, there was general consternation with this co-

It's Pitiful

lossal breach of security, the system was hastily shut down and the next twelve hours were spent heroically changing everyone's password. The question was how could this have happened? Let me explain.

To simplify the organization of the initial CTSS system, a design decision had been made to have each user at a terminal associated with his or her own directory of files. Moreover, the system itself was organized as a kind of quasi-

proceeded to cajole me into letting the system directory be an exception so that more than one person at a time could be logged into it. They assured me that they would be careful to not make mistakes.

But of course a mistake was made. A software design decision in the standard system text editor was overlooked. It was assumed that the editor would only be used by one user at a time working in one directory so that a temporary file could

and replace the previous ad hoc systems such as CTSS. It started as a cooperative effort among Project MAC of MIT, the Bell Telephone Laboratories, and the Computer Department of General Electric, later acquired by Honeywell. In our expansiveness of purpose we took on a long list of innovations.

Among the most important ones were the following: First, we introduced into the processor hardware the mechanisms for paging and segmentation along with a careful scheme for access control. Second, we introduced an idea for rings of protection around the supervisor software. Third, we planned from the start that the system would be composed of interchangeable multiple processors, memory modules, and so forth. And fourth, we made the decision to implement nearly all of the system in the newly defined compiler language, PL/I.

Let me share a few of my observations about the Multics experience. The novel hardware we had commissioned meant that the system had to be built from the ground up so that we had an immense task on our hands.

The decision to use a compiler to implement the system software was a good one, but what we did not appreciate was that new language PL/I presented us with two big difficulties: First, the language had constructs in it which were intrinsically complicated, and it required a learning period on the part of system programmers to learn to avoid them. Second, no one knew how to do a good job of implementing the compiler. Eventually we overcame these difficulties but it took precious time.

That Multics succeeded is remarkable, for it was the result of a cooperative effort of three highly independent organizations and had no administrative head. This meant decisions were made by persuasion and consensus. Consequently, it was difficult to reject weak ideas until considerable time and effort had been spent on them.

The Multics system did turn into

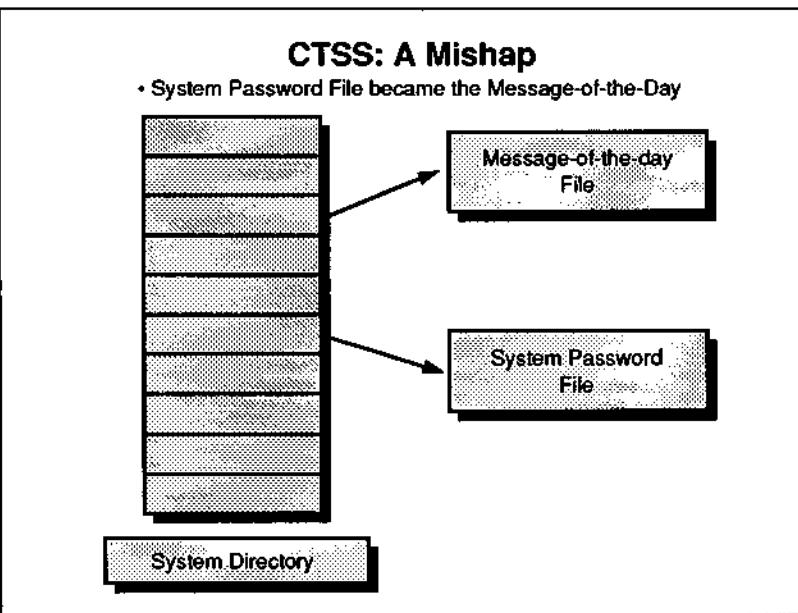


FIGURE 6
CTSS Is full of Surprises

user with its own directory that included a large number of supporting applications and files, including the message-of-the day and the password file. So far, so good. Normally a single-system programmer could login to the system directory and make any necessary changes. But the number of system programmers had grown to about a dozen in number, and, further, the system by then was being operated almost continuously so that the need to do live maintenance of the system files became essential. Not surprisingly, the system programmers saw the one-user-to-a-directory restriction as a big bottleneck for themselves. They thereupon

have the same name for all instantiations of the editor. But with two system programmers editing at the same time in the system directory, the editor temporary files became swapped and the disaster occurred.

One can draw two lessons from this: First, design bugs are often subtle and occur by evolution with early assumptions being forgotten as new features or uses are added to systems. Second, even skilled programmers make mistakes.

Multics

Let me turn now to the development of Multics [12]. I will be brief since the system has been documented well and there have already been two retrospective papers written [3, 4]. The Multics system was meant to do time-sharing "right"

a commercial product. Some of its major strengths were the virtual memory system, the file system, the attention to security, the ability to do online reconfiguration, and the information backup system for the file system.

And, as was also true with CTSS, many of the alumni of the Multics development have gone on to play important roles in the computing field [11].

A few more observations can be made about the ambitious Multics experience. In particular, we were misled by our earlier successes with previous systems such as CTSS, where we were able to build them "brick-by-brick," incrementally adding ideas to a large base of already working software.

We also were embarrassed by our inability to set and meet accurate schedules for completion of the different phases of the project. In retrospect, we should not have been, for we had never done anything like it before. However in many cases, our estimations should have been called guesses.

The Unix system [15] was a reaction to Multics. Even the name was a joke. Ken Thompson was part of the Bell Laboratories' Multics effort, and, frustrated with the attempts to bring a large system development under control, decided to start over. His strategy was clear—Start small and build up the ideas one by one as he saw how to implement them well. As we all know, Unix has evolved and become immensely successful as the system of choice for workstations. Still there are aspects of Multics that have never been replicated in Unix.

As a commercial product of Honeywell and Bull, Multics developed a loyal following. At the peak there were about 77 sites worldwide and even today many of the sites tenaciously continue for want of an alternative.

Sources of Complexity

The general problem with ambitious systems is complexity. Let me next try to abstract some of the

major causes. The most obvious complexity problems arise from scale. In particular, the larger the personnel required, the more levels of management there will be. We can see the problem even if we use simplistic calculations. Thus if we assume a fixed supervision ratio, for example six, the levels of management will grow as the logarithm of the personnel. The difficulty is that with more layers of management, the top-most layers become out of touch with the relevant bottom issues and the likelihood of random serendipitous communication decreases.

Another problem of organizations is that subordinates hate to report bad news, sometimes for fear of "being shot as the messenger" and at other times because they may have a different set of goals than the upper management.

And finally, large projects encourage specialization so that few team members understand all of the project. Misunderstandings and miscommunication begin, and soon a significant part of the project resources are spent fighting internal confusion. And, of course, mistakes occur.

My next category of complexity arises because of new design domains. The most vivid examples come from the world of physical systems, but software too is subject to the same problems, albeit often in more subtle ways.

Consider the destruction of the Tacoma Narrows Bridge, in Washington State, on November 7, 1940. The bridge had been proudly opened about four months earlier. Many of you have probably seen the amateur movie that was fortunately made of the collapse. What happened is that a strong but not unusual crosswind blew that day. Soon the roadbed, suspended by cables from the main span, began to vibrate like a reed, and the more it flexed, the better cross section it presented to the wind. The result was that the bridge tore itself apart as the oscillations became large and violent. What we had was a case of a

new design domain where the classic bridge builder, concerned with gravity-loaded structures, had entered into the realm of aeronautics. The result was a major mistake.

Next, let us look at the complexities that arise from human usage of computer systems. In using online systems that allow the sharing or exchanging of information—and here networked workstations clearly fall in this class—one is faced with a dilemma: If one places total trust in all other users, one is vulnerable to the antisocial behavior of any malicious user—consider the case of viruses.

But if one tries to be totally reclusive and isolated, one is not only bored, but one's information universe will cease to grow and be enhanced by interaction with others.

The result is that most of us operate in a complicated trade-off zone with various arrangements of trust and security mechanisms. Even such simple ideas as passwords are often a problem. They are a nuisance to remember, they can easily be compromised inadvertently, and they cannot be selectively revoked if shared. Privacy and security issues

(!@#%ing)

are particularly difficult to deal with since responsibilities are often split among users, managers, and vendors.

Worse yet, there is no way to simply "look" at a system and determine what the privacy and security implications are. It is no wonder mistakes occur all the time in this area.

One of the consequences of using computer systems is that increasingly information is being kept on-line in central storage devices. Computer storage devices have become remarkably reliable—except when they break—and that is the rub. Even the most experienced computer user can find him- or herself lulled into a false sense of security by the almost perfect operation of today's devices. The problem is compounded by the attitude of vendors, not unlike the initial attitude of the automobile industry toward safety, where inevitable disk failure is treated as a negative issue that dampens sales.

What is needed is constant vigilance against a long list of "what ifs": hardware failure, human slips, vandalism, theft, fire, earthquakes, long-term media failure, and even

the loss of institutional memories concerning recovery procedures. And as long as some individuals have to "learn the hard way," mistakes will continue to be made.

A further complication in discussing risk or reliability is that there is not a good language with which to carry on a dialog. Statistics are as often misapplied as they are misunderstood. We also get absurd absolutes such as "the Strategic Defense Initiative will produce a perfect unsaturable shield against nuclear attack" [14] or "it is impossible for the reactor to overheat." The problem is that we always have had risks in our lives, we never have been very good at discussing them, and with computers we now have a lot of new sources.

Another source of complexity arises with rapid change, change which is often driven by technology improvements. A result is that changes in procedures or usage occur and new vulnerabilities can arise. For example, in the area of telephone networks, the economies and efficiencies of fiber optic cables compared to copper wire are rapidly causing major upgrades and replacements in the national telephone plant. Because one fiber cable can carry at a reasonable cost the equivalent traffic of thousands of copper wires, fiber is quickly replacing copper. As a result, a transformation is likely to occur where network links become sparser over a given area and multiply interconnected nodes become less connected.

The difficulty is that there is reduced redundancy and a much higher vulnerability to isolated accidents. In the Chicago area not long ago there was a fire at a fiber optics switching center that caused a loss of service to a huge number of customers for several weeks. More recently, in New York City there was a shutdown of the financial exchanges for several hours because of a single mishap with a backhoe in New Jersey. Obviously in both instances, efficiency had gotten ahead of robustness.

The last source of complexity that I will single out arises from the frailty of human users when forced to deal with the multiplicity of technologies in modern life. In a little more than a century, there has been an awesome progression of technological changes from telephones and electricity, through automobiles, movies and radio—I will not even try to complete the list since we all know it well. The overall consequence has been to produce vast changes in our life-styles, and we see these changes even happening today. Consider the changes in the television editing styles that have occurred over a few decades, the impact of viewgraph overhead projectors on college classrooms, and the way we now do our banking with automatic teller machines. And the progression of life-style changes continues at a seemingly more rapid pace with word processing, answering machines, facsimile machines, and electronic mail.

One consequence of the many life-style changes is that some individuals feel stressed and overstimulated by the plethora of inputs. The natural defense is to increasingly depend on others to act as information filters. But the combination of stressful life-styles and insulation from original data will inevitably lead to more confusion and mistakes.

Conclusions

Most of this talk has been directed toward trying to persuade you that failures in complex, ambitious systems are inevitable. However, I would be remiss if I did not address ways to resolve the problem. Unfortunately, the list I can offer is rather short but worthy of brief review.

First, it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties and as we have seen, creating mistakes. My definition of elegance is the achievement of a given functionality with a minimum of mechanism and a maximum of clarity.

Second, the value of metaphors should not be underestimated. Metaphors have the virtue of an expected behavior that is understood by all. Unnecessary communication and misunderstandings are reduced. Learning and education are quicker. In effect, metaphors are a way of internalizing and abstracting concepts allowing one's thinking to be on a higher plane and low-level mistakes to be avoided.

Third, use of constrained languages for design or synthesis is a powerful methodology. By not allowing a programmer or designer to express irrelevant ideas, the domain of possible errors becomes far more limited.

Fourth, one must try to anticipate both errors of human usage and of hardware failure and properly develop the necessary contingency paths. This process of playing "what if" is not as easy as it may sound, since the need to attach likelihoods of occurrence to events and to address issues of the independence of failures is implicit.

Fifth, it should be assumed in the design of a system, that it will have to be repaired or modified. The overall effect will be a much more robust system, where there is a high degree of functional modularity and structure, and repairs can be made easily.

Sixth, and last, on a large project, one of the best investments that can be made is the cross education of the team so that nearly everyone knows more than he or she needs to know. Clearly, with educational redundancy, the team is more resilient to unexpected tragedies or departures. But in addition, the increased awareness of team members can help catch global or systemic mistakes early. It really is a case of "more heads are better than one."

Finally, I have touched on many different themes in this talk but I will single out three: First, the evolution of technology supports a rich future for ambitious visions and dreams that will inevitably involve

complex systems. Second, one must always try to learn from past mistakes, but at the same time be alert to the possibility that new circumstances require new solutions. And third, one must remember that ambitious systems demand a defensive philosophy of design and implementation. In other words, "Don't wonder if some mishap may happen, but rather ask what one will do about it when it does occur." ■

References

1. Brooks, F.P., Jr. No silver bullet. *IEEE Comput.* (Apr. 1987), 10-19.
2. Corbató, F.J. Sensitive issues in the design of multi-use systems. Unpublished lecture transcription of Feb. 1968, Project MAC Memo M-383.
3. Corbató, F.J., and Clingen, C.T. A managerial view of the Multics system development. In *Research Directions in Software Technology*, P. Wegner, Ed., M.I.T. Press, 1979. (Also published in *Tutorial: Software Management*, D.J. Reifer, Ed., IEEE Computer Society Press, 1979; Second Ed., 1981; Third Ed., 1986.)
4. Corbató, F.J., Clingen, C.T., and Saltzer, J.H. Multics: The first seven years. In *Proceedings of the SJCC* (May 1972), pp. 571-583.
5. Corbató, F.J., Daggett, M.M., and Daley, R.C. An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference* (May 1962).
6. Corbató, F.J., Daggett, M.M., Daley, R.C., Creasy, R.J., Hellwig, J.D., Orenstein, R.H., and Horn, L.K. *The Compatible Time-Sharing System: A Programmer's Guide*. M.I.T. Press, June 1963.
7. Corbató, F.J., and Vyssotsky, V.A. Introduction and overview of the Multics system. In *Proceedings FJCC* (1965).
8. Daley, R.C. and Neumann, P.G. A general-purpose file system for secondary storage. In *Proceedings FJCC* (1965).
9. David, E.E., Jr. and Fano, R.M. Some thoughts about the social implications of accessible computing. In *Proceedings FJCC* (1965).
10. Glaser, E.L., Couleur, J.F. and Oliver, G.A. System design of a computer for time-sharing applications. In *Proceedings FJCC* (1965).
11. Neumann, P.G., a Multics veteran.

has become a major contributor to the literature of computer-related risks. He is the editor of the widely-read network magazine "Risks-Forum," writes the "Inside Risks" column for the CACM, and periodically creates digests in the ACM Software Engineering Notes.

12. Organick, E.I. *The Multics System: An Examination of its Structure*. MIT Press, 1972.
13. Ossanna, J.F., Mikus, L. and Dunten, S.D. Communications and input-output switching in a Multiplex computing system. In *Proceedings FJCC* (1965).
14. Parnas, D.L. Software aspects of strategic defense systems. *Am. Sci.* (Nov. 1985). An excellent critique on the difficulties of producing software for large-scale systems.
15. Ritchie, D.M. and Thompson, K. The UNIX time-sharing system. *Commun. ACM* 17, 7 (July 1974), 365-375.
16. Vyssotsky, V.A., and Corbató, F.J. Structure of the Multics Supervisor. In *Proceedings FJCC*, 1965.

CR Categories and Subject Descriptors: C.2 [Computer Systems Organization]: Computer-Communication Networks; C.4 [Computer Systems Organization]: Performance of Systems; D.4 [Software]: Operating Systems; H.5 [Information Systems]: Information Interfaces and Presentation; K.2 [Computing Milieux]: History of Computing

General Terms: Design

About the Author:

FERNANDO J. CORBATÓ is Associate Head of Computer Science and Engineering at the Massachusetts Institute of Technology. **Author's Present Address:** Computer Science and Engineering Department, Room NE43-514, 545 Technology Square, Cambridge, MA 02139.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Turing On Computational Complexity Award and the Nature of Computer Science Lecture



In scientific work, the recognition by one's peers is one of the greatest rewards. In particular, an official recognition by the scientific community, as Richard Stearns and I are honored by the 1993 ACM Turing Award, is very satisfying and deeply appreciated.¹ Science is a great intellectual adventure and one of humankind's greatest achievements. Furthermore, a research career can be an exciting, rewarding and ennobling activity, particularly so if one is fortunate to participate in the creation of a completely new and very important science, as many scientists are. My road to computer science was not a direct one. Actually it looks more like a random walk, in retrospect, with the right intellectual stops to prepare me for work in computer science.

I was born in Latvia, which lost its independence during World War II and from which we had to flee because of heavy fighting at the end of World War II. After the war as a D.P. (displaced person) in Germany, I finished a superb Latvian high school in a D.P. camp staffed by elite refugee academics who conveyed their enthusiasm for knowledge, scholarship and particularly for science. I studied physics at the Philips University in Marburg and waited for a chance to emigrate to the United States. This chance came after about two-and-a-half years of studies. In the U.S. our sponsors were in Kansas City, and, after arriving there, I proceeded to the University of Kansas City (now part of the University of Missouri system). My two-plus years of study were judged to be the equivalent of a bachelor's degree, and I was accepted for graduate work and very generously awarded a fellowship. Since there was no graduate program in physics, I was advised (or told) to study mathematics, which had a graduate program. A year later I emerged with a master's degree in mathematics and with a far better appreciation of the power and beauty of mathematics. The California Institute of Technology accepted me for graduate work and from my record decided that I looked like "an applied mathematician" (which is probably what you get if you mix two years of European physics with a year of Kansas City mathematics, though I had never taken a course in applied mathematics). Since there was at that time no program in applied mathematics at Cal Tech, I was advised I would be perfectly happy studying pure mathematics.

This was good advice, and four years later, after one of the most stimulating intellectual periods in my life, I had earned my Ph.D. in mathematics with a dissertation in lattice theory and a minor in physics. Though I loved pure mathematics and was impressed by the beauty and power of mathematical abstractions, I felt some intellectual restlessness and a hope to find research problems with a more direct link to the world around us. Still, I followed my advisor's recommendation and accepted a faculty position in mathematics at Cornell University. During the second year at Cornell I was offered and accepted a summer job at General Electric Research Laboratory in Schenectady, N.Y., in their new information studies section headed by Dr. Richard Shuey. That summer was a sharp turning point in my scientific interests. At the GE Research Laboratory, I was caught up in the excitement about participating in the creation of a new science about information and computing. Computer science offered me the hoped-for research area with the right motivation, scope and excitement. One academic year later I joined the GE Research Laboratory as a research scientist. The following year Richard Stearns, a mathematics graduate student at Princeton, spent a summer at the Laboratory where we started our collaboration. After completing his Ph.D. at Princeton with a dissertation in game theory, Dick joined the Laboratory and we intensified our collaboration.

Our views of what kind of computer science we wanted to do were influenced by our backgrounds and the intensive study of the relevant literature we could find. We de-

lighted in Turing's 1936 paper [14] and were impressed by the elegance, crispness and simplicity of the undecidability results and basic recursive function theory. Turing's work supplied us with the necessary well-defined abstract computer model in our later work. I personally was deeply impressed with Shannon's communication theory [12]. Shannon's theory gave precise quantitative laws of how much information can be "reliably" transmitted over a noisy channel in terms of the channel capacity and the entropy of the information source. I loved physics for its beautifully precise laws that govern and explain the behavior of the physical world. In Shannon's work, for the first time, I saw precise quantitative laws that governed the behavior of the abstract entity of information. For an ex-physicist the idea that there could be quantitative laws governing such abstract entities as information and its transmission was surprising and immensely fascinating. Shannon had given a beautiful example of quantitative laws for information which by its nature is not directly constrained by physical laws. This raised the question whether there could be precise quantitative laws that govern the abstract process of computing, which again was not directly constrained by physical laws. Could there be quantitative laws that determine for each problem how much computing effort (work) is required for its solution and how to measure and determine it?

From these and other considerations grew our deep conviction that there must be quantitative laws that govern the behavior of information and computing. The results of this research effort were summarized in our first paper on this topic, which also named this new research area, "On the computational complexity of algorithms" [5]. To capture the quantitative behavior of the computing effort and to classify computations by their intrinsic computational complexity, which we were seeking, we needed a robust computing model and an intuitively satisfying classification of the complexity of problems. The Turing machine was ideally suited for the computer model, and we modified it to the multi-tape version. To classify computations (or problems) we introduced the key concept of a complexity class in terms of the Turing machines with bounded computational resources. A complexity class, for example, C_{n^2} in our original notation, consists of all problems whose instances of length n can be solved in n^2 steps on a multi-tape Turing machine. In contemporary notation, $C_{n^2} = \text{TIME}[n^2]$.

Today, complexity classes are central objects of study, and many results and problems in complexity theory are expressed in terms of complexity classes.

A considerable part of our early work on complexity theory was dedicated to showing that we had defined a meaningful classification of problems according to their computational difficulty and deriving results about it. We showed that our classification was robust and was not essentially altered by minor changes in the model and that the complexity classification indeed captured the intuitive ideas about the complexity of numbers and functions. We explored how computation speed changed by going from one-tape to multi-tape machines and even to multi-dimensional tapes and derived bounds for these "speed-

¹The Turing Award Lecture by co-recipient Richard Stearns will appear in the November issue of *Communications of the ACM*.

ups." Somewhat later, Manuel Blum, in his Ph.D. dissertation at MIT [1], developed an axiomatic theory of computational complexity and, among many other results, showed that all complexity measures are recursively related. Our speed-up results were special cases of this relationship. For us it was a delight to meet Manuel while he was writing his dissertation and to exchange ideas about computational complexity. Similarly, we were impressed and influenced by H. Yamada's work on real-time computations in his dissertation at the University of Pennsylvania [15] under the supervision of Robert McNaughton. We also proved Hierarchy Theorems that asserted that a slight increase in computation time (bounds) permits solution of new problems. More explicitly: if $T(n)$ and $U(n)$ are "nice" functions and

$$\lim_{n \rightarrow \infty} \frac{T(n)^2}{U(n)} = 0$$

then complexity class $\text{TIME}[T(n)]$ is properly contained in $\text{TIME}[U(n)]$. These results showed that there are problems with very sharp, intrinsic computational complexity bounds. No matter what method and computational algorithm was used, the problem solution required, say n^2 , operation for problem instance of size n . Blum in his dissertation showed that this is not the case for all problems and that there can exist exotic problems with less sharply defined bounds.

To relate our classification of the real numbers by their computation complexity to the classical concepts, we showed that all algebraic numbers are in the low complexity class $\text{TIME}[n^2]$ and found, to our surprise, some transcendental numbers that were real-time computable (i.e., they were in $\text{TIME}[n]$). Since we could not prove that any irrational algebraic numbers were in $\text{TIME}[n]$, we conjectured that all real-time computable numbers are either rational or transcendental. This is still an open problem 30 years later and only gradually did we realize its mathematical depth and the profound consequences its proof would have in mathematics.

Toward the end of the introduction of our first paper on complexity theory [5], we state: "The final section is devoted to open questions and problem areas. It is our conviction that numbers and functions have an intrinsic computational nature according to which they can be classified, as shown in this paper, and that there is a good opportunity here for further research." Indeed there was! We had opened a new computer science area of research and given it a name.

At the GE Research Laboratory, Phil Louis joined us to explore tape- (or memory-) bounded computations that yielded many interesting results and established computational space as another major computational resource measure [7, 13]. We showed that all context-free languages could be recognized on $(\log n)^2$ -tape. This result led Savitch [10] to his elegant result about the relation between deterministic and nondeterministic tape-bounded computations: for "nice" functions $F(n)$, $\text{NTAPE}[F(n)]$ is contained in $\text{TAPE}[F(n)^2]$.

Our colleague at the Laboratory, Daniel Younger [16], showed that context-free languages were contained in

$\text{TIME}[n^3]$. Soon many others joined the exploration of the complexity of computation, and computational complexity theory grew into a major research area with deep and interesting results and some of the most notorious open problems in computer science.

Looking at all of computer science and its history, I am very impressed by the scientific and technological achievements, and they far exceed what I had expected. Computer science has grown into an important science with rich intellectual achievements, an impressive arsenal of practical results and exciting future challenges. Equally impressive are the unprecedented technological developments in computing power and communication capacity that have amplified the scientific achievements and have given computing and computer science a central role in our scientific, intellectual and commercial activities.

I personally believe that computer science is not only a rapidly maturing science, but that it is more. Computer science differs so basically from the other sciences that it has to be viewed as a new species among the sciences, and it must be so understood. Computer science deals with information, its creation and processing, and with the systems that perform it, much of which is not directly restrained and governed by physical laws. Thus computer science is laying the foundations and developing the research paradigms and scientific methods for the exploration of the world of information and intellectual processes that are not directly governed by physical laws. This is what sets it apart from the other sciences and what we vaguely perceived and found fascinating in our early exploration of computational complexity.

One of the defining characteristics of computer science is the immense difference in scale of the phenomena computer science deals with. From the individual bits of programs and data in the computers to billions of operations per second on this information by the highly complex machines, their operating systems and the various languages in which the problems are described, the scale changes through many orders of magnitude. Donald Knuth² puts it nicely:

Computer Science and Engineering is a field that attracts a different kind of thinker. I believe that one who is a natural computer scientist thinks algorithmically. Such people are especially good at dealing with situations where different rules apply in different cases; they are individuals who can rapidly change levels of abstraction, simultaneously seeing things "in the large" and "in the small."

The computer scientist has to create many levels of abstractions to deal with these problems. One has to create intellectual tools to conceive, design, control, program, and reason about the most complicated of human creations. Furthermore, this has to be done with unprecedented precision. The underlying hardware that executes the computations are universal machines and therefore they are chaotic systems: the slightest change in their instructions or data can result in arbitrarily large differences in the results. This, as we well know, is an inherent prop-

²Personal communication, March 10, 1992 letter.

erty of universal computing devices (and theory makes clear that giving up universality imposes a very high price). Thus computer scientists are blessed with a universal device which can be instructed to perform any computation and simulate in principle any physical process (as described by our current laws of physics), but which is therefore chaotic and must be controlled with unprecedented precision. This is achieved by the successive layers of implemented abstraction wrapped around the chaotic universal machines that help to bridge the many orders of magnitude in the scale of things.

It is also this universality of the computing devices that gives the computing paradigm its immense power and scope. During various periods people have used the conceptualizations of their newest devices to try to understand and explain how nature and humans function. Thus our current heavy reliance on computer concepts and computer simulations for various phenomena has been compared to the use of the explanatory role of steam-driven devices, gears and latches, or clocks.

The universality of digital computers and the ever-increasing computing power give the computing paradigm a different and a very central role in all of our intellectual activities. The digital computer is a universal device and can perform in principle any computation (assuming the Church-Turing thesis, it captures all computations) and, in particular, any mathematical procedure in an axiomatized formal system. Thus in principle the full power of mathematical reasoning, which has been civilization's primary scientific tool, can be embodied in our computers from numerical computations and simulation of physical processes to symbolic computations and logical reasoning to theorem proving. This universality and the power of modern computers are indeed very encompassing of our intellectual activities and growing in scope and power.

Clearly, computer science is not a physical science; still, very often it is assumed that it will show strong similarities to physical sciences and may have similar research paradigms in regard to theory and experiments. The failure of computer science to conform to the paradigms of physical sciences is often interpreted as immaturity of computer science. This is not the case, since theory and experiments in computer science play a different role than in physical sciences. For a more detailed contrasting of the research paradigms in physics and computer science, see [4].

Even a brief look at research topics in computer science reveals the new relation between theory and experiments. For example, the design and analysis of algorithms is a central theme in theoretical computer science. Methods are developed for their design, measures are defined for various computational resources, trade-offs between different resources are explored, and upper- and lower-resource bounds are proved for the solutions of various problems. Similarly, theory creates methodologies, logics and various semantic models to help design programs, to reason about programs, to prove their correctness, and to guide the design of new programming languages. Theories develop models, measures and methods to explore and optimize VLSI designs, and to try to conceptualize

techniques to design efficient computer and communications systems.

Thinking about the previously mentioned (and other) theoretical work in computer science, one is led to the very clear conclusion that theories do not compete with each other for which better explains the fundamental nature of information. Nor are new theories developed to reconcile theory with experimental results that reveal unexplained anomalies or new, unexpected phenomena as in physics. In computer science there is no history of critical experiments that decide between the validity of various theories, as there are in physical sciences.

The basic, underlying mathematical model of digital computing is not seriously challenged by theory or experiments. The ultimate limits of *effective* computing, imposed by the theory of computing, are well understood and accepted. There is a strong effort to define and prove the *feasible* limits of computation, but even here the basic model of computation is not questioned. The key effort is to prove that certain computations cannot be done in given resource bounds, well illustrated by the $P = NP?$ question. One should note that the solution of this problem could have broad implications. For example, it could give proof of what encryption procedures are safe under what attacks and for how long. It could also lead to a deeper understanding of the limits of human-computer reasoning power. In general, the "separation" problems, that is the questions if $P \neq NP \neq PSPACE \neq EXPTIME \neq NEXPTIME \neq EXPSPACE$? are among the most important open problems in theoretical computer science. But there are no experiments, physical or computational, which could resolve these problems, again emphasizing the different scientific nature of computer science.

In computer science, results of theory are judged by the insights they reveal about the mathematical nature of various models of computing and/or by their utility to the practice of computing and their ease of applicability. Do the models conceptualize and capture the aspects computer scientists are interested in, do they yield insights in design problems, do they aid reasoning and communication about relevant problems? In the design and analysis of algorithms, which is a central theme in theoretical computer science, the measures of performance are well defined, and results can be compared quite easily in some of these measures (which may or may not fully reflect their performance on typical problems). Experiments with algorithms are used to test implementations and compare their "practical" performance on the subsets of problems deemed important.

Similarly, an inspection of the experimental work and systems building in computer science reveals a different pattern than in physical sciences. Such work deals with performance measurements, evaluation of design methodologies, testing of new architectures, and above all, testing feasibility by building systems to do what has never been done before.

Systems building, hardware and software, is the defining characteristic of applied and/or experimental work in computer science (though experimental is not meant in the old sense). This has the consequence that computer

*Looking at all of computer science and its history,
I am very impressed by the scientific
and technological achievements,
and they far exceed what I had expected.*

science advances are often demonstrated and documented by a dramatic demonstration rather than a dramatic experiment as in physical sciences. It is the role of the demo to show the possibility or feasibility to do what was thought to be impossible or not feasible. It is often that the (ideas and concepts tested in the) dramatic demos influence the research agenda in computer science.

This is reflected in the battle cry of the young computer scientists, "demo or die," which is starting to rival the older "publish or perish," which is still valid advice, but should be replaced by "publish in refereed journals or perish."

From the preceding observations we can see that theory and experiments in computer science are contributing to the design of algorithms and computing systems that execute them, that computer science is concentrating more on the *how* than the *what*, which is more the focal point of physical sciences. In general the *how* is associated with engineering, but computer science is not a subfield of engineering. Computer science is indeed an independent new science, but it is intertwined and permeated with engineering concerns and considerations. In many ways, the science and engineering aspects in computer science are much closer than in many other disciplines. To quote Fred Brooks [2] about programming:

The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and re-work, so readily capable of realizing grand conceptual structures. (. . . later, this very tractability has its own problems.)

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

Webster's dictionary defines engineering as "the application of scientific principles to practical ends as the design, construction, and operation of efficient and economical structures, equipment and systems." By this definition, much of computer science activity can be viewed as engineering or at least the search for those scientific principles which can be applied "to practical ends, design, construction, . . ." But again, keeping in mind Brooks' quote and reflecting on the scope of computer science and engineering activities, we see that the engineering in our field has different characteristics than the

more classical practice of engineering. Many of the engineering problems in computer science are not constrained by physical laws, and they demand the creation of new engineering paradigms and methodology.

As observed earlier, computer science work is permeated by concepts of efficiency and search for optimality. The "how" motivation of computer science brings engineering concepts into the science, and we should take pride in this nearness of our science to applicability.

Somewhat facetiously, but with a grain of truth in it, we can say that computer science is the engineering of mathematics (or mathematical processes). In these terms we see very strongly that it is a new form of engineering.

I am deeply convinced that we should not try to draw a sharp line between computer science and engineering and that any attempt to separate them is counterproductive.

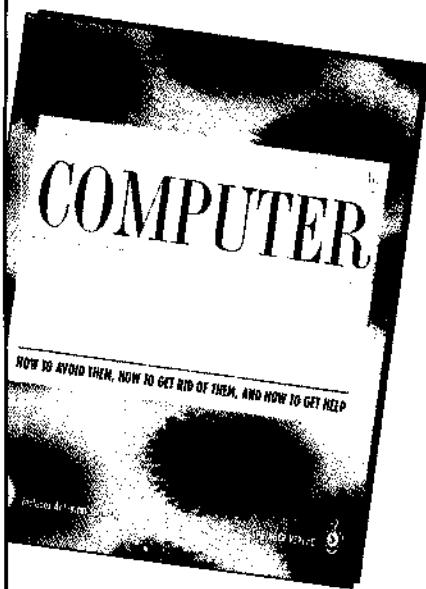
At the same time, I am convinced that computer science already has made and has a tremendous potential to make contributions to the understanding of our physical and intellectual world. The computing paradigm, supported by ever more powerful universal computing devices, motivates and permits the exploration and simulation of physical and intellectual processes and even assesses their power and limitations.

Already Warren McCullach in 1964 [8] had a vision of "Experimental epistemology, the study how knowledge is embodied in the brains and may be embodied in machines." John McCarthy [9] states less modestly, "The study of AI may lead to a mathematical metaepistemological analogous to metamathematics—to a study of the relations between a knower's rule for accepting evidence and the world in which he is embedded. This study could result in mathematical theorems about whether certain intellectual strategies can lead to the discovery of certain facts about the world. I think that this possibility will eventually revolutionize philosophy."

The computing paradigm has permitted the clarification of the concept of randomness by means of Kolmogorov complexity of finite and infinite sequences [6]. Again, the universality of the computing model was essential to prove the validity of these concepts.

Computational complexity considerations have refined the concepts of randomness relative to the intended applications. It has been shown that what is (acceptable or passes as) random depends on the computing power in the application. Still there remain deep open problems in this area about physical processes and computing. The Kolmogorov random strings are not computable, in a very strong sense; no Turing machine can print a Kolmogorov random string longer than its size (description). Does a

THE MOST UP-TO-DATE HANDBOOK ON COMPUTER VIRUSES



ROBERT SLADE'S GUIDE TO COMPUTER VIRUSES

How to Avoid Them, How to Get Rid of Them,
and How to Get Help

As society comes to rely more heavily on computers, the importance of safeguarding data from computer viruses should be of concern to all computer users. But how bad is the virus problem today? How bad will it become? If you find yourself bewildered, **Robert Slade's Guide to Computer Viruses** is a book you must read!

Written by a key figure in the virus protection community, this comprehensive book covers everything from the basics to detailed information on the most virulent and the newest viruses known. It also includes a complete review of the major antivirus software available. As a computer user, you will learn valuable guidelines on how to minimize the risk of infection, as well as how to access the latest information through electronic bulletin boards and news groups. This complete book will provide all the information you need to make informed decisions to protect your system, regardless of the platform you are using. Packaged with the book are five antivirus software programs to help you get started quickly.

1994/480 pp., 19 illus./Softcover \$29.95
Includes 3.5" diskette
ISBN 0-387-94311-0



Three Easy Ways to Order

CALL Toll-Free 1-800-SPRINGER (NJ call 201-348-4033) or FAX 201-348-4505. Please mention S965.

WRITE to Springer-Verlag New York, Inc.,
Attn: J. Jeng, 175 Fifth Avenue, Dept. S965,
New York, NY 10010-7858.

VISIT your local bookstore.

Payment can be made by check, purchase order, or credit card. Please enclose \$2.50 for shipping (\$1.00 each additional book) & add appropriate sales tax if you reside in CA, IL, MA, NJ, NY, PA, TX, VA, and VT. Canadian residents please add 7% GST. Foreign residents include \$10.00 airmail charge.

Remember...your 30-day return privilege is always guaranteed!
10/94 Reference #S965



Springer-Verlag New York

Circle # 11 on Reader Service Card

corresponding law (theorem?) hold for all physical systems? Can small physical systems produce long Kolmogorov random strings, or better yet, can a finite physical system (properly formulated with the needed energy inflow without adding randomness?) produce unbounded Kolmogorov random sequence? If so, then indeed physical processes are not fully capturable by computer simulation.

Very recently, computer science motivation has led to the study of interactive proofs and proof checking, revealing unexpected power of randomization and interaction between prover and verifier [11]. These results show that with very few questions about a long proof a verifier can be convinced with arbitrarily high probability that there is a correct proof without ever seeing the whole proof. These and related results have given fundamental new insights about the nature of mathematical proofs and are indeed metamathematical results.

Recursive function theory, originally motivated by Gödel's incompleteness results, classified what is and is not effectively computable, thus clearly showing the power and limitations of formal mathematical reasoning. Complexity theory is currently struggling to determine what is and is not feasibly computable. In this effort the $P = NP?$ problem is the best known open problem in this struggle, but by far not the only open separation problem. When the $P = NP?$ and other separation problems are resolved and deeper insights are gained about the limits of the feasibly computable, the computing paradigm and complexity theory may allow us to better understand the power and limitations of the human-computer reasoning. I believe that computer science has the potential to give deep new insights and quantitative understanding of the computing paradigm and our intellectual processes and thus, just maybe, a possibility to grasp the limits of the knowable.

Acknowledgments

The views expressed here have been deeply influenced by the author's participation in the National Research Council study resulting in the report, *Computing the Future: A Broader Agenda for Computer Science and Engineering* [3] and by discussions with colleagues at Cornell University and at the Max Planck Institut für Informatik in Saarbrücken, Germany. Particularly influential have been Robert Constable, Fred Schneider, and Richard Zippel. The importance of demos in computer science was emphasized by Constable and the importance of the immense differences in the scale of phenomena in computer science was eloquently explained by Zippel and compared to the still badly understood phenomena of turbulence in fluid dynamics, where the wide range of scales is contributing to the difficulty of the problem. ■

References

1. Blum, M. A machine independent theory of the complexity of recursive functions. *J. ACM* 14 (1967), 322-336.
2. Brooks, F.P. Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, Mass., 1975.
3. Hartmanis, J. and Lin, H., Eds. *Computing the Future: A Broader Agenda for Computer Science and Engineering*. National Academy Press, Washington, D.C., 1992.
4. Hartmanis, J. Some observations about the nature of com-

- puter science. In *Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, Vol. 761. Springer-Verlag, 1993, 1–12.
5. Hartmanis, J. and Stearns, R.E. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 177 (1965), 285–306.
 6. Li, M. and Vitanyi, P.M.B. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, Heidelberg, Germany, 1993.
 7. Lewis, P.M., Stearns, R.E., and Hartmanis, J. Memory bounds for the recognition for context-free and context-sensitive languages. In *Proceedings of IEEE Sixth Annual Symposium on Switching Circuit Theory and Logical Design*. (1965), pp. 191–202.
 8. McCullagh, W.S. A historical introduction to the postulational foundations of experimental epistemology. In *Cross-Cultural Understanding: Epistemology in Anthropology*. F.C.S. Northrop, and H.H. Livingston, Eds., Harper and Row, New York, 1964.
 9. McCarthy, J. Mathematical logic and artificial intelligence. In *The Artificial Intelligence Debate*. S.R. Graubard, Ed., MIT Press, Cambridge, Mass., 1988.
 10. Savitch, W.J. Relationship between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4 (1970), 177–192.
 11. Shamir, A. IP = PSPACE. *J. ACM* 39 (1992), 869–877.
 12. Shannon, C. The mathematical theory communication. *Bell System Tech. J.* 27 (1948), 379–656.
 13. Stearns, R.E., Hartmanis, J., and Lewis, P.M. Hierarchies of memory limited computations. In *Proceedings of IEEE Sixth Annual Symposium on Switching Circuit Theory and Logical Design*. (1965), pp. 179–190.
 14. Turing, A.M. On computable numbers with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, series 2, 42 (1936), 230–265.
 15. Yamada, H. Real-time computation and recursive functions not real-time computable. *IEEE Trans. Elec. Comput.* II, 6 (1962), 753–760.
 16. Younger, D.H. Recognition and parsing of context-free languages in time n^k . *Information and Control* 10, 2 (1967), 189–208.

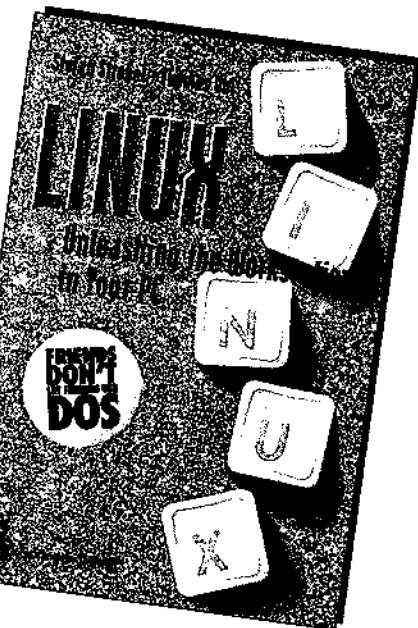
About the Author:

JURIS HARTMANIS is the Walter R. Read Professor of Engineering in the department of computer science at Cornell University. Current research interests include theory of computing, computational complexity, and structural complexity. **Author's Present Address:** Department of Computer Science, Cornell University, 5149 Upson Hall, Ithaca, NY 14853; email: jh@cs.cornell.edu

This research was supported in part by National Science Foundation grant #CCR-9123730 and by the Alexander von Humboldt Foundation and the Max Planck Institut für Informatik in Saarbrücken, Germany.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

THE ONE HANDBOOK FOR LINUX USERS



STEFAN STROBEL & THOMAS UHL

LINUX - UNLEASHING THE WORKSTATION IN YOUR PC

Basics, Installation and Practical Use

Linux has emerged as a viable alternative to commercial UNIX systems, with its ability to turn a 386/486-PC into a UNIX workstation with performance characteristics comparable to a RISC workstation. As the definitive guide to Linux, this book introduces the concepts and features of Linux and explains how to install and configure the system. Moreover, it describes the features and services of the Internet which have been instrumental in the rapid development and wide distribution of Linux. This book focuses on the Linux graphical interface, its network capability and extended tools. Using the book, readers can get started quickly with Linux and begin to explore a wide range of shareware applications that are available for the system.

1994/238 pp., 50 illus./Softcover \$29.95
ISBN 0-387-58077-8

Three Easy Ways to Order

CALL Toll-Free 1-800-SPRINGER (NJ call 201-348-4033) or FAX 201-348-4505. Please mention S966.

WRITE to Springer-Verlag New York, Inc.,

Attn: J. Jeng, 175 Fifth Avenue, Dept. S966,

New York, NY 10010-7858.

VISIT your local bookstore.

Payment can be made by check, purchase order, or credit card. Please enclose \$2.50 for shipping (\$1.00 each additional book) & add appropriate sales tax if you reside in CA, IL, MA, NJ, NY, PA, TX, VA, and VT. Canadian residents please add 7% GST. Foreign residents include \$10.00 airmail charge.

Remember...your 30-day return privilege is always guaranteed!

10/94 Reference #S966



Springer-Verlag New York

Circle # 11 on Reader Service Card

1974 ACM Turing Award Lecture

[The Turing Award citation read by Bernard A. Galler, chairman of the 1974 Turing Award Committee, on the presentation of this lecture on November 11 at the ACM Annual Conference in San Diego.]

The A.M. Turing Award of the ACM is presented annually by the ACM to an individual selected for his contributions of a technical nature made to the computing community. In particular, these contributions should have had significant influence on a major segment of the computer field.

"The 1974 A.M. Turing Award is presented to Professor Donald E. Knuth of Stanford University for a number of major contributions to the analysis of algorithms and the design of programming languages, and in particular for his most significant contributions to the 'art of computer programming' through his series of well-known books. The collections of techniques, algorithms and relevant theory in these books have served as a focal point for developing curricula and as an organizing influence on computer science."

Such a formal statement cannot put into proper perspective the role which Don Knuth has been playing in computer science, and in the computer industry as a whole. It has been my experi-

ence with respect to the first recipient of the Turing Award, Professor Alan J. Perlis, that at every meeting in which he participates he manages to provide the insight into the problems being discussed that becomes the focal point of discussion for the rest of the meeting. In a very similar way, the vocabulary, the examples, the algorithms, and the insight that Don Knuth has provided in his excellent collection of books and papers have begun to find their way into a great many discussions in almost every area of the field. This does not happen easily. As every author knows, even a single volume requires a great deal of careful organization and hard work. All the more must we appreciate the clear view and the patience and energy which Knuth must have had to plan seven volumes and to set about implementing his plan so carefully and thoroughly.

It is significant that this award and the others that he has been receiving are being given to him after three volumes of his work have been published. We are clearly ready to signal to everyone our appreciation of Don Knuth for his dedication and his contributions to our discipline. I am very pleased to have chaired the Committee that has chosen Don Knuth to receive the 1974 A.M. Turing Award of the ACM.

Computer Programming as an Art

by Donald E. Knuth

When *Communications of the ACM* began publication in 1959, the members of ACM's Editorial Board made the following remark as they described the purposes of ACM's periodicals [2]: "If computer programming is to become an important part of computer research and development, a transition of programming from an art to a disciplined science must be effected." Such a goal has been a continually recurring theme during the ensuing years; for example, we read in 1970 of the "first steps toward transforming the art of programming into a science" [26]. Meanwhile we have actually succeeded in making our discipline a science, and in a remarkably simple way: merely by deciding to call it "computer science."

Implicit in these remarks is the notion that there is something undesirable about an area of human activity

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address, Computer Science Department, Stanford University, Stanford, CA 94305

that is classified as an "art"; it has to be a Science before it has any real stature. On the other hand, I have been working for more than 12 years on a series of books called "*The Art of Computer Programming*." People frequently ask me why I picked such a title; and in fact some people apparently don't believe that I really did so, since I've seen at least one bibliographic reference to some books called "*The Act of Computer Programming*."

In this talk I shall try to explain why I think "Art" is the appropriate word. I will discuss what it means for something to be an art, in contrast to being a science; I will try to examine whether arts are good things or bad things; and I will try to show that a proper viewpoint of the subject will help us all to improve the quality of what we are now doing.

One of the first times I was ever asked about the title of my books was in 1966, during the last previous ACM national meeting held in Southern California. This was before any of the books were published, and I recall having lunch with a friend at the convention hotel. He knew how conceited I was, already at that

time, so he asked if I was going to call my books "An Introduction to Don Knuth." I replied that, on the contrary, I was naming the books after *him*. His name: Art Evans. (*The Art of Computer Programming*, in person.)

From this story we can conclude that the word "art" has more than one meaning. In fact, one of the nicest things about the word is that it is used in many different senses, each of which is quite appropriate in connection with computer programming. While preparing this talk, I went to the library to find out what people have written about the word "art" through the years; and after spending several fascinating days in the stacks, I came to the conclusion that "art" must be one of the most interesting words in the English language.

The Arts of Old

If we go back to Latin roots, we find *ars*, *artis* meaning "skill." It is perhaps significant that the corresponding Greek word was *τέχνη*, the root of both "technology" and "technique."

Nowadays when someone speaks of "art" you probably think first of "fine arts" such as painting and sculpture, but before the twentieth century the word was generally used in quite a different sense. Since this older meaning of "art" still survives in many idioms, especially when we are contrasting art with science, I would like to spend the next few minutes talking about art in its classical sense.

In medieval times, the first universities were established to teach the seven so-called "liberal arts," namely grammar, rhetoric, logic, arithmetic, geometry, music, and astronomy. Note that this is quite different from the curriculum of today's liberal arts colleges, and that at least three of the original seven liberal arts are important components of computer science. At that time, an "art" meant something devised by man's intellect, as opposed to activities derived from nature or instinct; "liberal" arts were liberated or free, in contrast to manual arts such as plowing (cf. [6]). During the middle ages the word "art" by itself usually meant logic [4], which usually meant the study of syllogisms.

Science vs. Art

The word "science" seems to have been used for many years in about the same sense as "art"; for example, people spoke also of the seven liberal sciences, which were the same as the seven liberal arts [1]. Duns Scotus in the thirteenth century called logic "the Science of Sciences, and the Art of Arts" (cf. [12, p. 34f]). As civilization and learning developed, the words

took on more and more independent meanings, "science" being used to stand for knowledge, and "art" for the application of knowledge. Thus, the science of astronomy was the basis for the art of navigation. The situation was almost exactly like the way in which we now distinguish between "science" and "engineering."

Many authors wrote about the relationship between art and science in the nineteenth century, and I believe the best discussion was given by John Stuart Mill. He said the following things, among others, in 1843 [28]:

Several sciences are often necessary to form the groundwork of a single art. Such is the complication of human affairs, that to enable one thing to be *done*, it is often requisite to *know* the nature and properties of many things . . . Art in general consists of the truths of Science, arranged in the most convenient order for practice, instead of the order which is the most convenient for thought. Science groups and arranges its truths so as to enable us to take in at one view as much as possible of the general order of the universe. Art . . . brings together from parts of the field of science most remote from one another, the truths relating to the production of the different and heterogeneous conditions necessary to each effect which the exigencies of practical life require.

As I was looking up these things about the meanings of "art," I found that authors have been calling for a transition from art to science for at least two centuries. For example, the preface to a textbook on mineralogy, written in 1784, said the following [17]: "Previous to the year 1780, mineralogy, though tolerably understood by many as an Art, could scarce be deemed a Science."

According to most dictionaries "science" means knowledge that has been logically arranged and systematized in the form of general "laws." The advantage of science is that it saves us from the need to think things through in each individual case; we can turn our thoughts to higher-level concepts. As John Ruskin wrote in 1853 [32]: "The work of science is to substitute facts for appearances, and demonstrations for impressions."

It seems to me that if the authors I studied were writing today, they would agree with the following characterization: Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it. Since the notion of an algorithm or a computer program provides us with an extremely useful test for the depth of our knowledge about any given subject, the process of going from an art to a science means that we learn how to automate something.

Artificial intelligence has been making significant progress, yet there is a huge gap between what computers can do in the foreseeable future and what ordinary people can do. The mysterious insights that

people have when speaking, listening, creating, and even when they are programming, are still beyond the reach of science; nearly everything we do is still an art.

From this standpoint it is certainly desirable to make computer programming a science, and we have indeed come a long way in the 15 years since the publication of the remarks I quoted at the beginning of this talk. Fifteen years ago computer programming was so badly understood that hardly anyone even *thought* about proving programs correct; we just fiddled with a program until we "knew" it worked. At that time we didn't even know how to express the *concept* that a program was correct, in any rigorous way. It is only in recent years that we have been learning about the processes of abstraction by which programs are written and understood; and this new knowledge about programming is currently producing great payoffs in practice, even though few programs are actually proved correct with complete rigor, since we are beginning to understand the principles of program structure. The point is that when we write programs today, we know that we could in principle construct formal proofs of their correctness if we really wanted to, now that we understand how such proofs are formulated. This scientific basis is resulting in programs that are significantly more reliable than those we wrote in former days when intuition was the only basis of correctness.

The field of "automatic programming" is one of the major areas of artificial intelligence research today. Its proponents would love to be able to give a lecture entitled "Computer Programming as an Artifact" (meaning that programming has become merely a relic of bygone days), because their aim is to create machines that write programs better than we can, given only the problem specification. Personally I don't think such a goal will ever be completely attained, but I do think that their research is extremely important, because everything we learn about programming helps us to improve our own artistry. In this sense we should continually be striving to transform *every* art into a science: in the process, we advance the art.

Science and Art

Our discussion indicates that computer programming is by now *both* a science and an art, and that the two aspects nicely complement each other. Apparently most authors who examine such a question come to this same conclusion, that their subject is both a science and an art, whatever their subject is (cf. [25]). I found a book about elementary photography, written in 1893, which stated that "the development of the photographic image is both an art and a science" [13]. In fact, when I first

picked up a dictionary in order to study the words "art" and "science," I happened to glance at the editor's preface, which began by saying, "The making of a dictionary is both a science and an art." The editor of Funk & Wagnall's dictionary [27] observed that the painstaking accumulation and classification of data about words has a scientific character, while a well-chosen phrasing of definitions demands the ability to write with economy and precision: "The science without the art is likely to be ineffective; the art without the science is certain to be inaccurate."

When preparing this talk I looked through the card catalog at Stanford library to see how other people have been using the words "art" and "science" in the titles of their books. This turned out to be quite interesting.

For example, I found two books entitled *The Art of Playing the Piano* [5, 15], and others called *The Science of Pianoforte Technique* [10], *The Science of Pianoforte Practice* [30]. There is also a book called *The Art of Piano Playing: A Scientific Approach* [22].

Then I found a nice little book entitled *The Gentle Art of Mathematics* [31], which made me somewhat sad that I can't honestly describe computer programming as a "gentle art."

I had known for several years about a book called *The Art of Computation*, published in San Francisco, 1879, by a man named C. Frusher Howard [14]. This was a book on practical business arithmetic that had sold over 400,000 copies in various editions by 1890. I was amused to read the preface, since it shows that Howard's philosophy and the intent of his title were quite different from mine; he wrote: "A knowledge of the Science of Number is of minor importance; skill in the Art of Reckoning is absolutely indispensable."

Several books mention both science and art in their titles, notably *The Science of Being and Art of Living* by Maharishi Mahesh Yogi [24]. There is also a book called *The Art of Scientific Discovery* [11], which analyzes how some of the great discoveries of science were made.

So much for the word "art" in its classical meaning. Actually when I chose the title of my books, I wasn't thinking primarily of art in this sense, I was thinking more of its current connotations. Probably the most interesting book which turned up in my search was a fairly recent work by Robert E. Mueller called *The Science of Art* [29]. Of all the books I've mentioned, Mueller's comes closest to expressing what I want to make the central theme of my talk today, in terms of real artistry as we now understand the term. He observes: "It was once thought that the imaginative

outlook of the artist was death for the scientist. And the logic of science seemed to spell doom to all possible artistic flights of fancy." He goes on to explore the advantages which actually do result from a synthesis of science and art.

A scientific approach is generally characterized by the words logical, systematic, impersonal, calm, rational, while an artistic approach is characterized by the words aesthetic, creative, humanitarian, anxious, irrational. It seems to me that both of these apparently contradictory approaches have great value with respect to computer programming.

Emma Lehmer wrote in 1956 that she had found coding to be "an exacting science as well as an intriguing art" [23]. H.S.M. Coxeter remarked in 1957 that he sometimes felt "more like an artist than a scientist" [7]. This was at the time C.P. Snow was beginning to voice his alarm at the growing polarization between "two cultures" of educated people [34, 35]. He pointed out that we need to combine scientific and artistic values if we are to make real progress.

Works of Art

When I'm sitting in an audience listening to a long lecture, my attention usually starts to wane at about this point in the hour. So I wonder, are you getting a little tired of my harangue about "science" and "art"? I really hope that you'll be able to listen carefully to the rest of this, anyway, because now comes the part about which I feel most deeply.

When I speak about computer programming as an art, I am thinking primarily of it as an art *form*, in an aesthetic sense. The chief goal of my work as educator and author is to help people learn how to write *beautiful programs*. It is for this reason I was especially pleased to learn recently [32] that my books actually appear in the Fine Arts Library at Cornell University. (However, the three volumes apparently sit there neatly on the shelf, without being used, so I'm afraid the librarians may have made a mistake by interpreting my title literally.)

My feeling is that when we prepare a program, it can be like composing poetry or music; as Andrei Ershov has said [9], programming can give us both intellectual and emotional satisfaction, because it is a real achievement to master complexity and to establish a system of consistent rules.

Furthermore when we read other people's programs, we can recognize some of them as genuine works of art. I can still remember the great thrill it was for me to read the listing of Stan Poley's SOAP II assembly

program in 1958; you probably think I'm crazy, and styles have certainly changed greatly since then, but at the time it meant a great deal to me to see how elegant a system program could be, especially by comparison with the heavy-handed coding found in other listings I had been studying at the same time. The possibility of writing beautiful programs, even in assembly language, is what got me hooked on programming in the first place.

Some programs are elegant, some are exquisite, some are sparkling. My claim is that it is possible to write *grand programs*, *noble programs*, truly *magnificent ones*!

Taste and Style

The idea of *style* in programming is now coming to the forefront at last, and I hope that most of you have seen the excellent little book on *Elements of Programming Style* by Kernighan and Plauger [16]. In this connection it is most important for us all to remember that there is no one "best" style; everybody has his own preferences, and it is a mistake to try to force people into an unnatural mold. We often hear the saying, "I don't know anything about art, but I know what I like." The important thing is that you really *like* the style you are using; it should be the best way you prefer to express yourself.

Edsger Dijkstra stressed this point in the preface to his *Short Introduction to the Art of Programming* [8]:

It is my purpose to transmit the importance of good taste and style in programming, [but] the specific elements of style presented serve only to illustrate what benefits can be derived from "style" in general. In this respect I feel akin to the teacher of composition at a conservatory: He does not teach his pupils how to compose a particular symphony, he must help his pupils to find their own style and must explain to them what is implied by this. (It has been this analogy that made me talk about "The Art of Programming.")

Now we must ask ourselves, What is good style, and what is bad style? We should not be too rigid about this in judging other people's work. The early nineteenth-century philosopher Jeremy Bentham put it this way [3, Bk. 3, Ch. 1]:

Judges of elegance and taste consider themselves as benefactors to the human race, whilst they are really only the interrupters of their pleasure . . . There is no taste which deserves the epithet *good*, unless it be the taste for such employments which, to the pleasure actually produced by them, conjoin some contingent or future utility: there is no taste which deserves to be characterized as bad, unless it be a taste for some occupation which has a mischievous tendency.

When we apply our own prejudices to "reform" someone else's taste, we may be unconsciously denying him some entirely legitimate pleasure. That's why I don't

condemn a lot of things programmers do, even though I would never enjoy doing them myself. The important thing is that they are creating something *they* feel is beautiful.

In the passage I just quoted, Bentham does give us some advice about certain principles of aesthetics which are better than others, namely the "utility" of the result. We have some freedom in setting up our personal standards of beauty, but it is especially nice when the things we regard as beautiful are also regarded by other people as useful. I must confess that I really enjoy writing computer programs; and I especially enjoy writing programs which do the greatest good, in some sense.

There are many senses in which a program can be "good," of course. In the first place, it's especially good to have a program that works correctly. Secondly it is often good to have a program that won't be hard to change, when the time for adaptation arises. Both of these goals are achieved when the program is easily readable and understandable to a person who knows the appropriate language.

Another important way for a production program to be good is for it to interact gracefully with its users, especially when recovering from human errors in the input data. It's a real art to compose meaningful error messages or to design flexible input formats which are not error-prone.

Another important aspect of program quality is the efficiency with which the computer's resources are actually being used. I am sorry to say that many people nowadays are condemning program efficiency, telling us that it is in bad taste. The reason for this is that we are now experiencing a reaction from the time when efficiency was the only reputable criterion of goodness, and programmers in the past have tended to be so preoccupied with efficiency that they have produced needlessly complicated code; the result of this unnecessary complexity has been that net efficiency has gone down, due to difficulties of debugging and maintenance.

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

We shouldn't be penny wise and pound foolish, nor should we always think of efficiency in terms of so many percent gained or lost in total running time or space. When we buy a car, many of us are almost oblivious to a difference of \$50 or \$100 in its price, while we might make a special trip to a particular store in order to buy a 50¢ item for only 25¢. My point

is that there is a time and place for efficiency; I have discussed its proper role in my paper on structured programming, which appears in the current issue of *Computing Surveys* [21].

Less Facilities: More Enjoyment

One rather curious thing I've noticed about aesthetic satisfaction is that our pleasure is significantly enhanced when we accomplish something with limited tools. For example, the program of which I personally am most pleased and proud is a compiler I once wrote for a primitive minicomputer which had only 4096 words of memory, 16 bits per word. It makes a person feel like a real virtuoso to achieve something under such severe restrictions.

A similar phenomenon occurs in many other contexts. For example, people often seem to fall in love with their Volkswagens but rarely with their Lincoln Continentals (which presumably run much better). When I learned programming, it was a popular pastime to do as much as possible with programs that fit on only a single punched card. I suppose it's this same phenomenon that makes APL enthusiasts relish their "one-liners." When we teach programming nowadays, it is a curious fact that we rarely capture the heart of a student for computer science until he has taken a course which allows "hands on" experience with a minicomputer. The use of our large-scale machines with their fancy operating systems and languages doesn't really seem to engender any love for programming, at least not at first.

It's not obvious how to apply this principle to increase programmers' enjoyment of their work. Surely programmers would groan if their manager suddenly announced that the new machine will have only half as much memory as the old. And I don't think anybody, even the most dedicated "programming artists," can be expected to welcome such a prospect, since nobody likes to lose facilities unnecessarily. Another example may help to clarify the situation: Film-makers strongly resisted the introduction of talking pictures in the 1920's because they were justly proud of the way they could convey words without sound. Similarly, a true programming artist might well resent the introduction of more powerful equipment; today's mass storage devices tend to spoil much of the beauty of our old tape sorting methods. But today's film makers don't want to go back to silent films, not because they're lazy but because they know it is quite possible to make beautiful movies using the improved technology. The form of their art has changed, but there is still plenty of room for artistry.

How did they develop their skill? The best film

makers through the years usually seem to have learned their art in comparatively primitive circumstances, often in other countries with a limited movie industry. And in recent years the most important things we have been learning about programming seem to have originated with people who did not have access to very large computers. The moral of this story, it seems to me, is that we should make use of the idea of limited resources in our own education. We can all benefit by doing occasional "toy" programs, when artificial restrictions are set up, so that we are forced to push our abilities to the limit. We shouldn't live in the lap of luxury all the time, since that tends to make us lethargic. The art of tackling miniproblems with all our energy will sharpen our talents for the real problems, and the experience will help us to get more pleasure from our accomplishments on less restricted equipment.

In a similar vein, we shouldn't shy away from "art for art's sake"; we shouldn't feel guilty about programs that are just for fun. I once got a great kick out of writing a one-statement ALGOL program that invoked an innerproduct procedure in such an unusual way that it calculated the m th prime number, instead of an innerproduct [19]. Some years ago the students at Stanford were excited about finding the shortest FORTRAN program which prints itself out, in the sense that the program's output is identical to its own source text. The same problem was considered for many other languages. I don't think it was a waste of time for them to work on this; nor would Jeremy Bentham, whom I quoted earlier, deny the "utility" of such pastimes [3, Bk. 3, Ch. 1]. "On the contrary," he wrote, "there is nothing, the utility of which is more incontestable. To what shall the character of utility be ascribed, if not to that which is a source of pleasure?"

Providing Beautiful Tools

Another characteristic of modern art is its emphasis on creativity. It seems that many artists these days couldn't care less about creating beautiful things; only the novelty of an idea is important. I'm not recommending that computer programming should be like modern art in this sense, but it does lead me to an observation that I think is important. Sometimes we are assigned to a programming task which is almost hopelessly dull, giving us no outlet whatsoever for any creativity; and at such times a person might well come to me and say, "So programming is beautiful? It's all very well for you to declaim that I should take pleasure in creating elegant and charming programs, but how am I supposed to make this mess into a work of art?"

Well, it's true, not all programming tasks are going to be fun. Consider the "trapped housewife," who has to clean off the same table every day: there's not room for creativity or artistry in every situation. But even in such cases, there is a way to make a big improvement: it is still a pleasure to do routine jobs if we have beautiful things to work with. For example, a person will really enjoy wiping off the dining room table, day after day, if it is a beautifully designed table made from some fine quality hardwood.

Therefore I want to address my closing remarks to the system programmers and the machine designers who produce the systems that the rest of us must work with. Please, give us tools that are a pleasure to use, especially for our routine assignments, instead of providing something we have to fight with. Please, give us tools that encourage us to write better programs, by enhancing our pleasure when we do so.

It's very hard for me to convince college freshmen that programming is beautiful, when the first thing I have to tell them is how to punch "slash slash job equals so-and-so." Even job control languages can be designed so that they are a pleasure to use, instead of being strictly functional.

Computer hardware designers can make their machines much more pleasant to use, for example by providing floating-point arithmetic which satisfies simple mathematical laws. The facilities presently available on most machines make the job of rigorous error analysis hopelessly difficult, but properly designed operations would encourage numerical analysts to provide better subroutines which have certified accuracy (cf. [20, p. 204]).

Let's consider also what software designers can do. One of the best ways to keep up the spirits of a system user is to provide routines that he can interact with. We shouldn't make systems too automatic, so that the action always goes on behind the scenes; we ought to give the programmer-user a chance to direct his creativity into useful channels. One thing all programmers have in common is that they enjoy working with machines; so let's keep them in the loop. Some tasks are best done by machine, while others are best done by human insight; and a properly designed system will find the right balance. (I have been trying to avoid misdirected automation for many years, cf. [18].)

Program measurement tools make a good case in point. For years, programmers have been unaware of how the real costs of computing are distributed in their programs. Experience indicates that nearly everybody has the wrong idea about the real bottlenecks in his

programs; it is no wonder that attempts at efficiency go awry so often, when a programmer is never given a breakdown of costs according to the lines of code he has written. His job is something like that of a newly married couple who try to plan a balanced budget without knowing how much the individual items like food, shelter, and clothing will cost. All that we have been giving programmers is an optimizing compiler, which mysteriously does something to the programs it translates but which never explains what it does. Fortunately we are now finally seeing the appearance of systems which give the user credit for some intelligence; they automatically provide instrumentation of programs and appropriate feedback about the real costs. These experimental systems have been a huge success, because they produce measurable improvements, and especially because they are fun to use, so I am confident that it is only a matter of time before the use of such systems is standard operating procedure. My paper in *Computing Surveys* [21] discusses this further, and presents some ideas for other ways in which an appropriate interactive routine can enhance the satisfaction of user programmers.

Language designers also have an obligation to provide languages that encourage good style, since we all know that style is strongly influenced by the language in which it is expressed. The present surge of interest in structured programming has revealed that none of our existing languages is really ideal for dealing with program and data structure, nor is it clear what an ideal language should be. Therefore I look forward to many careful experiments in language design during the next few years.

Summary

To summarize: We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better. Therefore we can be glad that people who lecture at computer conferences speak about the *state of the Art*.

References

1. Bailey, Nathan. *The Universal Etymological English Dictionary*. T. Cox, London, 1727. See "Art," "Liberal," and "Science."
2. Bauer, Walter F., Juncosa, Mario L., and Perlis, Alan J. ACM publication policies and plans. *J. ACM* 6 (Apr. 1959), 121-122.
3. Bentham, Jeremy. *The Rationale of Reward*. Trans. from *Théorie des peines et des récompenses*, 1811, by Richard Smith, J. & H. L. Hunt, London, 1825.

4. *The Century Dictionary and Cyclopedia* 1. The Century Co., New York, 1889.
5. Clementi, Muzio. *The Art of Playing the Piano*. Trans. from *L'art de jouer le pianoforte* by Max Vogrich. Schirmer, New York, 1898.
6. Colvin, Sidney. "Art." *Encyclopaedia Britannica*, eds 9, 11, 12, 13, 1875-1926.
7. Coxeter, H. S. M. Convocation address, Proc. 4th Canadian Math. Congress, 1957, pp. 8-10.
8. Dijkstra, Edsger W. EWD316: *A Short Introduction to the Art of Programming*. T. H. Eindhoven, The Netherlands, Aug. 1971.
9. Ershov, A. P. Aesthetics and the human factor in programming. *Comm. ACM* 15 (July 1972), 501-505.
10. Fielden, Thomas. *The Science of Pianoforte Technique*. Macmillan, London, 1927.
11. Gore, George. *The Art of Scientific Discovery*. Longmans, Green, London, 1878.
12. Hamilton, William. *Lectures on Logic* 1. Wm. Blackwood, Edinburgh, 1874.
13. Hodges, John A. Elementary Photography: *The "Amateur Photographer" Library* 7. London, 1893. Sixth ed, revised and enlarged, 1907, p. 58.
14. Howard, C. Frusher. Howard's *Art of Computation* and golden rule for equation of payments for schools, business colleges and self-culture C.F. Howard, San Francisco, 1879.
15. Hummel, J.N. *The Art of Playing the Piano Forte*. Boosey, London, 1827.
16. Kernighan B.W., and Plauger, P.J. *The Elements of Programming Style*. McGraw-Hill, New York, 1974.
17. Kirwan, Richard. *Elements of Mineralogy*. Elmsly, London, 1784.
18. Knuth, Donald E. Minimizing drum latency time. *J. ACM* 8 (Apr. 1961), 119-150.
19. Knuth, Donald E., and Merner, J.N. ALGOL 60 confidential. *Comm. ACM* 4 (June 1961), 268-272.
20. Knuth, Donald E. Seminumerical Algorithms: *The Art of Computer Programming* 2. Addison-Wesley, Reading, Mass., 1969.
21. Knuth, Donald E. Structured programming with go to statements. *Computing Surveys* 6 (Dec. 1974), pages in makeup.
22. Kochetvitsky, George. *The Art of Piano Playing: A Scientific Approach*. Summy-Birchard, Evanston, Ill., 1967.
23. Lehmer, Emma. Number theory on the SWAC. *Proc. Symp. Applied Math.* 6, Amer. Math. Soc. (1956), 103-108.
24. Mahesh Yogi, Maharishi. *The Science of Being and Art of Living*. Allen & Unwin, London, 1963.
25. Malevinsky, Moses L. *The Science of Playwriting*. Brentano's, New York, 1925.
26. Manna, Zohar, and Pnueli, Amir. Formalization of properties of functional programs. *J. ACM* 17 (July 1970), 555-569.
27. Marckwardt, Albert H. Preface to *Fink and Wagnall's Standard College Dictionary*. Harcourt, Brace & World, New York, 1963, vii.
28. Mill, John Stuart. *A System of Logic, Ratiocinative and Inductive*. London, 1843. The quotations are from the introduction, §2, and from Book 6, Chap. 11 (12 in later editions), §§.
29. Mueller, Robert E. *The Science of Art*. John Day, New York, 1967.
30. Parsons, Albert Ross. *The Science of Pianoforte Practice*. Schirmer, New York, 1886.
31. Pedoe, Daniel. *The Gentle Art of Mathematics*. English U. Press, London, 1953.
32. Ruskin, John. *The Stones of Venice* 3. London, 1853.
33. Salton, G.A. Personal communication, June 21, 1974.
34. Snow, C.P. The two cultures. *The New Statesman and Nation* 52 (Oct. 6, 1956), 413-414.
35. Snow, C.P. *The Two Cultures: and a Second Look*. Cambridge University Press, 1964.

COMBINATORICS, COMPLEXITY, AND RANDOMNESS

The 1985 Turing Award winner presents his perspective on the development of the field that has come to be called theoretical computer science.

RICHARD M. KARP

*This lecture is dedicated to the memory of my father,
Abraham Louis Karp.*

I am honored and pleased to be the recipient of this year's Turing Award. As satisfying as it is to receive such recognition, I find that my greatest satisfaction as a researcher has stemmed from doing the research itself, and from the friendships I have formed along the way. I would like to roam with you through my 25 years as a researcher in the field of combinatorial algorithms and computational complexity, and tell you about some of the concepts that have seemed important to me, and about some of the people who have inspired and influenced me.

BEGINNINGS

My entry into the computer field was rather accidental. Having graduated from Harvard College in 1955 with a degree in mathematics, I was confronted with a decision as to what to do next. Working for a living had little appeal, so graduate school was the obvious choice. One possibility was to pursue a career in mathematics, but the field was then in the heyday of its emphasis on abstraction and generality, and the concrete and applicable mathematics that I enjoyed the most seemed to be out of fashion.

And so, almost by default, I entered the Ph.D. program at the Harvard Computation Laboratory. Most of the topics that were to become the bread and butter of the computer science curriculum had not even been thought of then, and so I took an eclectic collection of courses: switching theory, numerical analysis, applied mathematics, probability and statistics, operations research, electronics, and mathematical linguistics. While the curriculum left much to be desired in depth and

coherence, there was a very special spirit in the air; we knew that we were witnessing the birth of a new scientific discipline centered on the computer. I discovered that I found beauty and elegance in the structure of algorithms, and that I had a knack for the discrete mathematics that formed the basis for the study of computers and computation. In short, I had stumbled more or less by accident into a field that was very much to my liking.

EASY AND HARD COMBINATORIAL PROBLEMS

Ever since those early days, I have had a special interest in combinatorial search problems—problems that can be likened to jigsaw puzzles where one has to assemble the parts of a structure in a particular way. Such problems involve searching through a finite, but extremely large, structured set of possible solutions, patterns, or arrangements, in order to find one that satisfies a stated set of conditions. Some examples of such problems are the placement and interconnection of components on an integrated circuit chip, the scheduling of the National Football League, and the routing of a fleet of school buses.

Within any one of these combinatorial puzzles lurks the possibility of a combinatorial explosion. Because of the vast, furiously growing number of possibilities that have to be searched through, a massive amount of computation may be encountered unless some subtlety is used in searching through the space of possible solutions. I'd like to begin the technical part of this talk by telling you about some of my first encounters with combinatorial explosions.

My first defeat at the hands of this phenomenon came soon after I joined the IBM Yorktown Heights Research Center in 1959. I was assigned to a group headed by J. P. Roth, a distinguished algebraic topolo-

gist who had made notable contributions to switching theory. Our group's mission was to create a computer program for the automatic synthesis of switching circuits. The input to the program was a set of Boolean formulas specifying how the outputs of the circuit were to depend on the inputs; the program was supposed to generate a circuit to do the job using a minimum number of logic gates. Figure 1 shows a circuit for the majority function of three variables; the output is high whenever at least two of the three variables x , y , and z are high.

The program we designed contained many elegant shortcuts and refinements, but its fundamental mechanism was simply to enumerate the possible circuits in order of increasing cost. The number of circuits that the program had to comb through grew at a furious rate as the number of input variables increased, and as a consequence, we could never progress beyond the solution of toy problems. Today, our optimism in even trying an enumerative approach may seem utterly naive, but we are not the only ones to have fallen into this trap; much of the work on automatic theorem proving over the past two decades has begun with an initial surge of excitement as toy problems were successfully solved, followed by disillusionment as the full seriousness of the combinatorial explosion phenomenon became apparent.

Around this same time, I began working on the traveling salesman problem with Michael Held of IBM. This problem takes its name from the situation of a salesman who wishes to visit all the cities in his territory, beginning and ending at his home city, while minimizing his total travel cost. In the special case where the cities are points in the plane and travel cost is equated with Euclidean distance, the problem is simply to find a polygon of minimum perimeter passing through all the cities (see Figure 2). A few years earlier, George Dantzig, Raymond Fulkerson, and Selmer Johnson at the Rand Corporation, using a mixture of manual and automatic computation, had succeeded in solving a 49-city problem, and we hoped to break their record.

Despite its innocent appearance, the traveling salesman problem has the potential for a combinatorial ex-

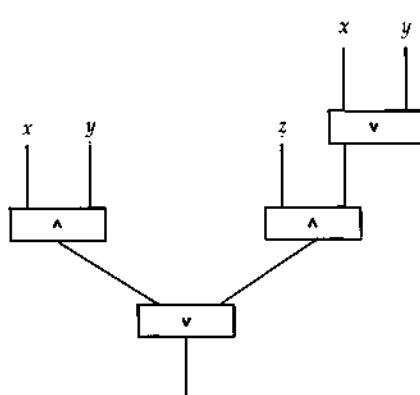


FIGURE 1. A Circuit for the Majority Function

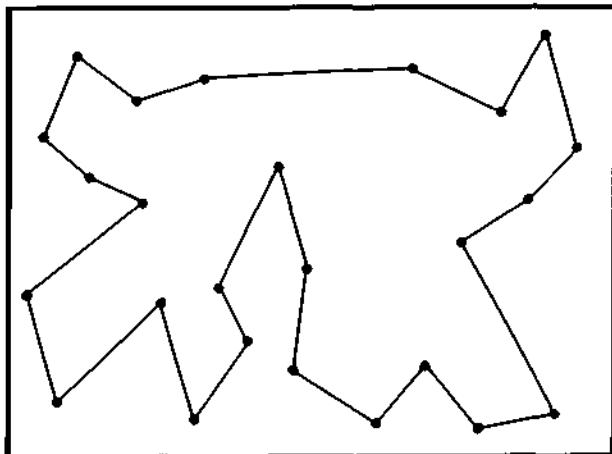


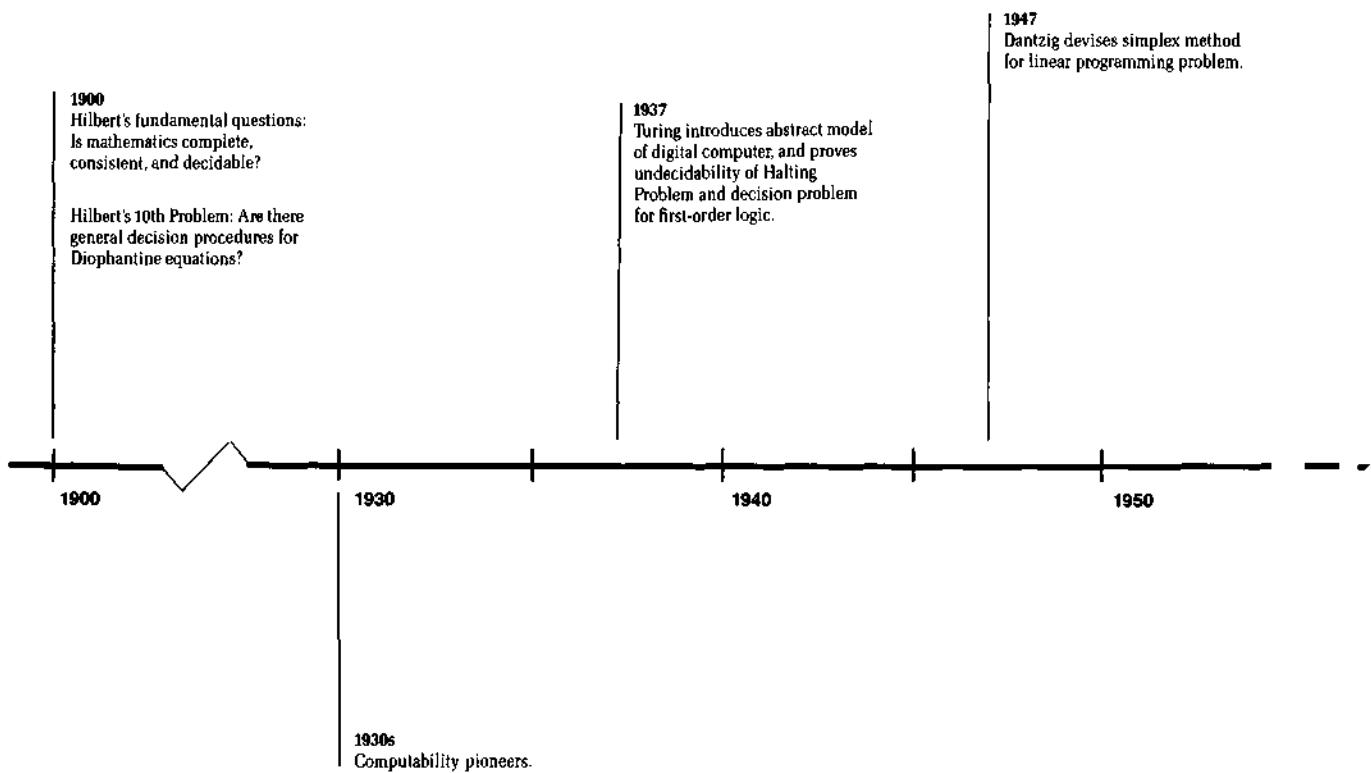
FIGURE 2. A Traveling Salesman Tour

plosion, since the number of possible tours through n cities in the plane is $(n - 1)!/2$, a very rapidly growing function of n . For example, when the number of cities is only 20, the time required for a brute-force enumeration of all possible tours, at the rate of a million tours per second, would be more than a thousand years.

Held and I tried a number of approaches to the traveling salesman problem. We began by rediscovering a shortcut based on dynamic programming that had originally been pointed out by Richard Bellman. The dynamic programming method reduced the search time to $n^2 2^n$, but this function also blows up explosively, and the method is limited in practice to problems with at most 16 cities. For a while, we gave up on the idea of solving the problem exactly, and experimented with local search methods that tend to yield good, but not optimal, tours. With these methods, one starts with a tour and repeatedly looks for local changes that will improve it. The process continues until a tour is found that cannot be improved by any such local change. Our local improvement methods were rather clumsy, and much better ones were later found by Shen Lin and Brian Kernighan at Bell Labs. Such quick-and-dirty methods are often quite useful in practice if a strictly optimal solution is not required, but one can never guarantee how well they will perform.

We then began to investigate branch-and-bound methods. Such methods are essentially enumerative in nature, but they gain efficiency by pruning away large parts of the space of possible solutions. This is done by computing a lower bound on the cost of every tour that includes certain links and fails to include certain others; if the lower bound is sufficiently large, it will follow that no such tour can be optimal. After a long series of unsuccessful experiments, Held and I stumbled upon a powerful method of computing lower bounds. This bounding technique allowed us to prune the search severely, so that we were able to solve problems with as many as 65 cities. I don't think any of my theoretical results have provided as great a thrill as the sight of the numbers pouring out of the computer on

THE DEVELOPMENT OF COMBINATORIAL OPTIMIZATION AND COMPUTATIONAL COMPLEXITY THEORY



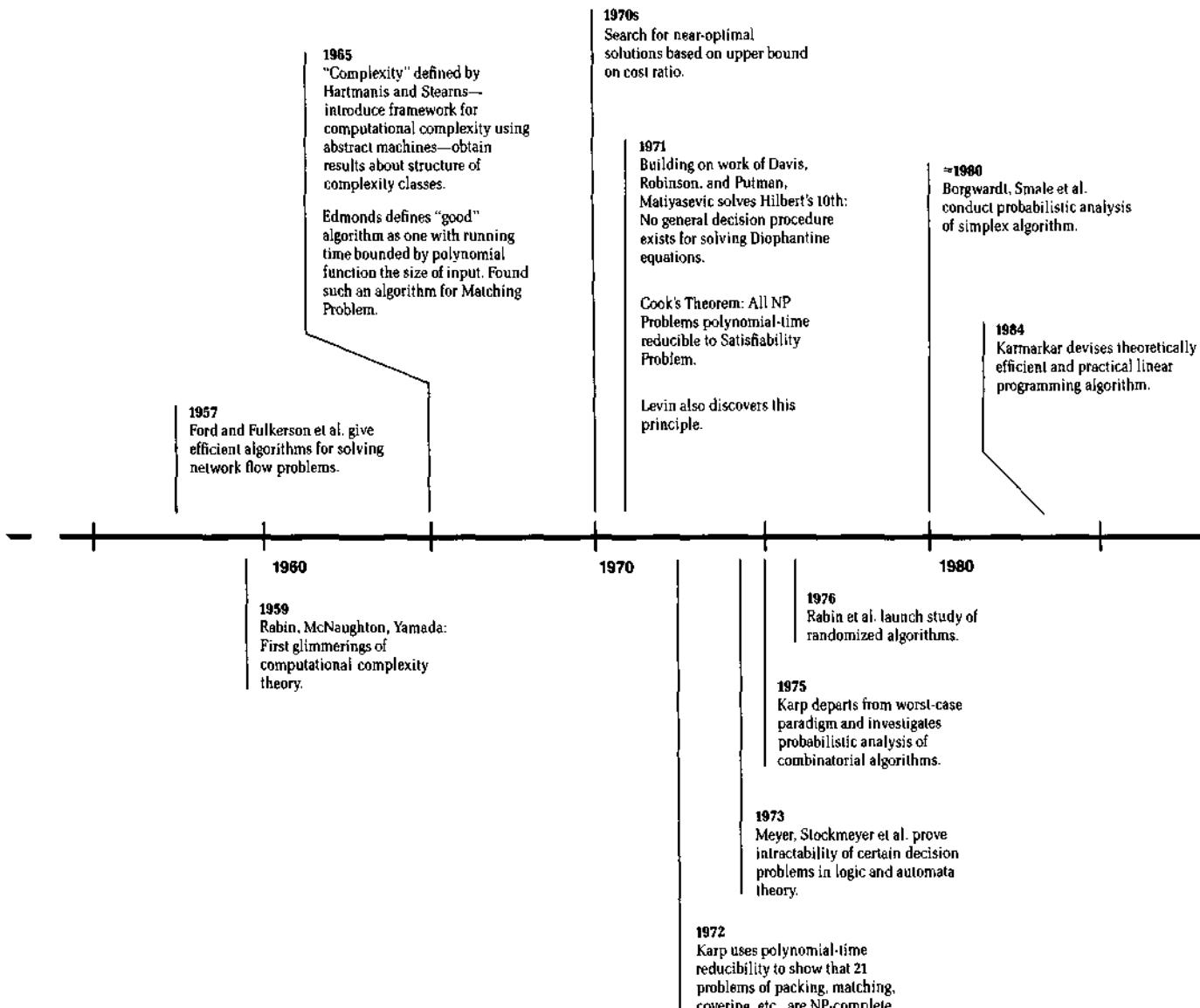
the night Held and I first tested our bounding method. Later we found out that our method was a variant of an old technique called Lagrangian relaxation, which is now used routinely for the construction of lower bounds within branch-and-bound methods.

For a brief time, our program was the world champion traveling-salesman-problem solver, but nowadays much more impressive programs exist. They are based on a technique called polyhedral combinatorics, which attempts to convert instances of the traveling salesman problem to very large linear programming problems. Such methods can solve problems with over 300 cities, but the approach does not completely eliminate combinatorial explosions, since the time required to solve a problem continues to grow exponentially as a function of the number of cities.

The traveling salesman problem has remained a fascinating enigma. A book of more than 400 pages has recently been published, covering much of what is

known about this elusive problem. Later, we will discuss the theory of NP-completeness, which provides evidence that the traveling salesman problem is inherently intractable, so that no amount of clever algorithm design can ever completely defeat the potential for combinatorial explosions that lurks within this problem.

During the early 1960s, the IBM Research Laboratory at Yorktown Heights had a superb group of combinatorial mathematicians, and under their tutelage, I learned important techniques for solving certain combinatorial problems without running into combinatorial explosions. For example, I became familiar with Dantzig's famous simplex algorithm for linear programming. The linear programming problem is to find the point on a polyhedron in a high-dimensional space that is closest to a given external hyperplane (a polyhedron is the generalization of a polygon in two-dimensional space or an ordinary polyhedral body in three-dimensional



space, and a hyperplane is the generalization of a line in the plane or a plane in three-dimensional space). The closest point to the hyperplane is always a corner point, or vertex, of the polyhedron (see Figure 3). In practice, the simplex method can be depended on to find the desired vertex very quickly.

I also learned the beautiful network flow theory of Lester Ford and Fulkerson. This theory is concerned with the rate at which a commodity, such as oil, gas, electricity, or bits of information, can be pumped through a network in which each link has a capacity that limits the rate at which it can transmit the commodity. Many combinatorial problems that at first sight seem to have no relation to commodities flowing through networks can be recast as network flow problems, and the theory enables such problems to be solved elegantly and efficiently using no arithmetic operations except addition and subtraction.

Let me illustrate this beautiful theory by sketching

the so-called Hungarian algorithm for solving a combinatorial optimization problem known as the marriage problem. This problem concerns a society consisting of n men and n women. The problem is to pair up the men and women in a one-to-one fashion at minimum cost, where a given cost is imputed to each pairing. These costs are given by an $n \times n$ matrix, in which each row corresponds to one of the men and each column to one of the women. In general, each pairing of the n men with the n women corresponds to a choice of n entries from the matrix, no two of which are in the same row or column; the cost of a pairing is the sum of the n entries that are chosen. The number of possible pairings is $n!$, a function that grows so rapidly that brute-force enumeration will be of little avail. Figure 4a shows a 3×3 example in which we see that the cost of pairing man 3 with woman 2 is equal to 9, the entry in the third row and second column of the given matrix.

The key observation underlying the Hungarian algo-

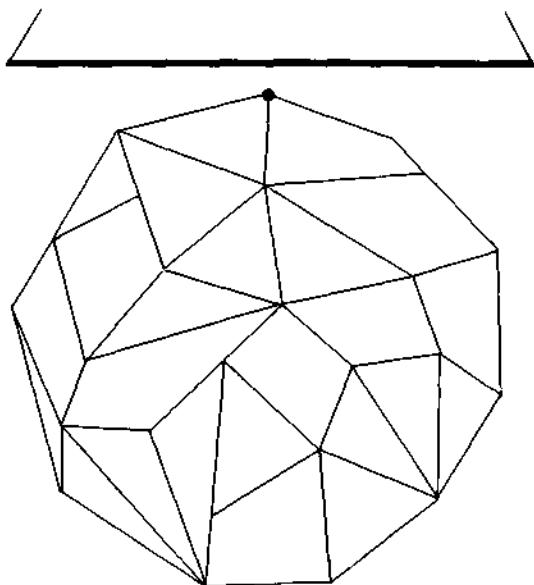


FIGURE 3. The Linear Programming Problem

rithm is that the problem remains unchanged if the same constant is subtracted from all the entries in one particular row of the matrix. Using this freedom to alter the matrix, the algorithm tries to create a matrix in which all the entries are nonnegative, so that every complete pairing has a nonnegative total cost, and in which there exists a complete pairing whose entries are all zero. Such a pairing is clearly optimal for the cost matrix that has been created, and it is optimal for the original cost matrix as well. In our 3×3 example, the algorithm starts by subtracting the least entry in each row from all the entries in that row, thereby creating a matrix in which each row contains at least one zero (Figure 4b). Then, to create a zero in each column, the algorithm subtracts, from all entries in each column that does not already contain a zero, the least entry in that column (Figure 4c). In this example, all the zeros in the resulting matrix lie in the first row or the third column; since a complete pairing contains only one entry from each row or column, it is not yet possible to find a complete pairing consisting entirely of zero entries. To create such a pairing, it is necessary to create a zero in the lower left part of the matrix. In this case, the algorithm creates such a zero by subtracting 1 from the first and second columns and adding 1 to the first row (Figure 4d). In the resulting nonnegative matrix, the three circled entries give a complete pairing of cost

$$(a) \begin{bmatrix} 3 & 4 & 2 \\ 8 & 9 & 1 \\ 7 & 9 & 5 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 2 & 0 \\ 7 & 8 & 0 \\ 2 & 4 & 0 \end{bmatrix} \quad (c) \begin{bmatrix} 0 & 0 & 0 \\ 6 & 6 & 0 \\ 1 & 2 & 0 \end{bmatrix} \quad (d) \begin{bmatrix} 0 & 0 & 1 \\ 5 & 5 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

FIGURE 4. An Instance of the Marriage Problem

zero, and this pairing is therefore optimal, both in the final matrix and in the original one.

This algorithm is far subtler and more efficient than brute-force enumeration. The time required for it to solve the marriage problem grows only as the third power of n , the number of rows and columns of the matrix, and as a consequence, it is possible to solve examples with thousands of rows and columns.

The generation of researchers who founded linear programming theory and network flow theory had a pragmatic attitude toward issues of computational complexity: An algorithm was considered efficient if it ran fast enough in practice, and it was not especially important to prove it was fast in all possible cases. In 1967 I noticed that the standard algorithm for solving certain network flow problems had a theoretical flaw, which caused it to run very slowly on certain contrived examples. I found that it was not difficult to correct the flaw, and I gave a talk about my result to the combinatorics seminar at Princeton. The Princeton people informed me that Jack Edmonds, a researcher at the National Bureau of Standards, had presented very similar results at the same seminar during the previous week.

As a result of this coincidence, Edmonds and I began to work together on the theoretical efficiency of network flow algorithms, and in due course, we produced a joint paper. But the main effect of our collaboration was to reinforce some ideas about computational complexity that I was already groping toward and that were to have a powerful influence on the future course of my research. Edmonds was a wonderful craftsman who had used ideas related to linear programming to develop amazing algorithms for a number of combinatorial problems. But, in addition to his skill at constructing algorithms, he was ahead of his contemporaries in another important respect: He had developed a clear and precise understanding of what it meant for an algorithm to be efficient. His papers expounded the point of view that an algorithm should be considered "good" if its running time is bounded by a polynomial function of the size of the input, rather than, say, by an exponential function. For example, according to Edmonds's concept, the Hungarian algorithm for the marriage problem is a good algorithm because its running time grows as the third power of the size of the input. But as far as we know there may be no good algorithm for the traveling salesman problem, because all the algorithms we have tried experience an exponential growth in their running time as a function of problem size. Edmonds's definition gave us a clear idea of how to define the boundary between easy and hard combinatorial problems and opened up for the first time, at least in my thinking, the possibility that we might someday come up with a theorem that would prove or disprove the conjecture that the traveling salesman problem is inherently intractable.

THE ROAD TO NP-COMPLETENESS

Along with the developments in the field of combinatorial algorithms, a second major stream of research was

gathering force during the 1960s—computational complexity theory. The foundations for this subject were laid in the 1930s by a group of logicians, including Alan Turing, who were concerned with the existence or nonexistence of automatic procedures for deciding whether mathematical statements were true or false.

Turing and the other pioneers of computability theory were the first to prove that certain well-defined mathematical problems were undecidable, that is, that, in principle, there could not exist an algorithm capable of solving all instances of such problems. The first example of such a problem was the Halting Problem, which is essentially a question about the debugging of computer programs. The input to the Halting Problem is a computer program, together with its input data; the problem is to decide whether the program will eventually halt. How could there fail to be an algorithm for such a well-defined problem? The difficulty arises because of the possibility of unbounded search. The obvious solution is simply to run the program until it halts. But at what point does it become logical to give up, to decide that the program isn't going to halt? There seems to be no way to set a limit on the amount of search needed. Using a technique called diagonalization, Turing constructed a proof that no algorithm exists that can successfully handle all instances of the Halting Problem.

Over the years, undecidable problems were found in almost every branch of mathematics. An example from number theory is the problem of solving Diophantine equations: Given a polynomial equation such as

$$4xy^2 + 2xy^2z^3 - 11x^3y^2z^2 = -1164,$$

is there a solution in integers? The problem of finding a general decision procedure for solving such Diophantine equations was first posed by David Hilbert in 1900, and it came to be known as Hilbert's Tenth Problem. The problem remained open until 1971, when it was proved that no such decision procedure can exist.

One of the fundamental tools used in demarcating the boundary between solvable and unsolvable problems is the concept of reducibility, which was first brought into prominence through the work of logician Emil Post. Problem A is said to be reducible to problem B if, given a subroutine capable of solving problem B, one can construct an algorithm to solve problem A. As an example, a landmark result is that the Halting Problem is reducible to Hilbert's Tenth Problem (see Figure 5). It follows that Hilbert's Tenth Problem must be undecidable, since otherwise we would be able to use this reduction to derive an algorithm for the Halting Problem, which is known to be undecidable. The concept of reducibility will come up again when we discuss NP-completeness and the P : NP problem.

Another important theme that complexity theory inherited from computability theory is the distinction between the ability to solve a problem and the ability to check a solution. Even though there is no general method to find a solution to a Diophantine equation, it

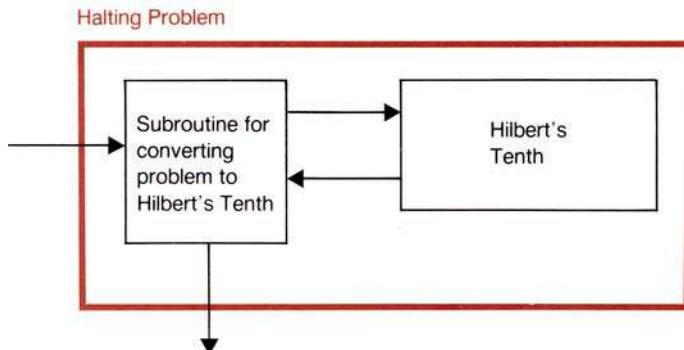


FIGURE 5. The Halting Problem Is Reducible to Hilbert's Tenth Problem

is easy to check a proposed solution. For example, to check whether $x = 3, y = 2, z = -1$ constitutes a solution to the Diophantine equation given above, one merely plugs in the given values and does a little arithmetic. As we will see later, the distinction between solving and checking is what the P : NP problem is all about.

Some of the most enduring branches of theoretical computer science have their origins in the abstract machines and other formalisms of computability theory. One of the most important of these branches is computational complexity theory. Instead of simply asking whether a problem is decidable at all, complexity theory asks how difficult it is to solve the problem. In other words, complexity theory is concerned with the capabilities of universal computing devices such as the Turing machine when restrictions are placed on their execution time or on the amount of memory they may use. The first glimmerings of complexity theory can be found in papers published in 1959 and 1960 by Michael Rabin and by Robert McNaughton and Hideo Yamada, but it is the 1965 paper by Juris Hartmanis and Richard Stearns that marks the beginning of the modern era of complexity theory. Using the Turing machine as their model of an abstract computer, Hartmanis and Stearns provided a precise definition of the "complexity class" consisting of all problems solvable in a number of steps bounded by some given function of the input length n . Adapting the diagonalization technique that Turing had used to prove the undecidability of the Halting Problem, they proved many interesting results about the structure of complexity classes. All of us who read their paper could not fail to realize that we now had a satisfactory formal framework for pursuing the questions that Edmonds had raised earlier in an intuitive fashion—questions about whether, for instance, the traveling salesman problem is solvable in polynomial time.

In that same year, I learned computability theory from a superb book by Hartley Rogers, who had been my teacher at Harvard. I remember wondering at the time whether the concept of reducibility, which was so central in computability theory, might also have a role to play in complexity theory, but I did not see how to make the connection. Around the same time, Michael Rabin, who was to receive the Turing Award in 1976,

was a visitor at the IBM Research Laboratory at Yorktown Heights, on leave from the Hebrew University in Jerusalem. We both happened to live in the same apartment building on the outskirts of New York City, and we fell into the habit of sharing the long commute to Yorktown Heights. Rabin is a profoundly original thinker and one of the founders of both automata theory and complexity theory, and through my daily discussions with him along the Sawmill River Parkway, I gained a much broader perspective on logic, computability theory, and the theory of abstract computing machines.

In 1968, perhaps influenced by the general social unrest that gripped the nation, I decided to move to the University of California at Berkeley, where the action was. The years at IBM had been crucial for my development as a scientist. The opportunity to work with such outstanding scientists as Alan Hoffman, Raymond Miller, Arnold Rosenberg, and Shmuel Winograd was simply priceless. My new circle of colleagues included Michael Harrison, a distinguished language theorist who had recruited me to Berkeley, Eugene Lawler, an expert on combinatorial optimization, Manuel Blum, a founder of complexity theory who has gone on to do outstanding work at the interface between number theory and cryptography, and Stephen Cook, whose work in complexity theory was to influence me so greatly a few years later. In the mathematics department, there were Julia Robinson, whose work on Hilbert's Tenth Problem was soon to bear fruit, Robert Solovay, a famous logician who later discovered an important randomized algorithm for testing whether a number is prime, and Steve Smale, whose ground-breaking work on the probabilistic analysis of linear programming algorithms was to influence me some years later. And across the Bay at Stanford were Dantzig, the father of linear programming, Donald Knuth, who founded the fields of data structures and analysis of algorithms, as well as Robert Tarjan, then a graduate student, and John Hopcroft, a sabbatical visitor from Cornell, who were brilliantly applying data structure techniques to the analysis of graph algorithms.

In 1971 Cook, who by then had moved to the University of Toronto, published his historic paper "On the Complexity of Theorem-Proving Procedures." Cook discussed the classes of problems that we now call P and NP, and introduced the concept that we now refer to as NP-completeness. Informally, the class P consists of all those problems that can be solved in polynomial time. Thus the marriage problem lies in P because the Hungarian algorithm solves an instance of size n in about n^3 steps, but the traveling salesman problem appears not to lie in P, since every known method of solving it requires exponential time. If we accept the premise that a computational problem is not tractable unless there is a polynomial-time algorithm to solve it, then all the tractable problems lie in P. The class NP consists of all those problems for which a proposed solution can be checked in polynomial time. For example, consider a version of the traveling sales-

man problem in which the input data consist of the distances between all pairs of cities, together with a "target number" T , and the task is to determine whether there exists a tour of length less than or equal to T . It appears to be extremely difficult to determine whether such a tour exists, but if a proposed tour is given to us, we can easily check whether its length is less than or equal to T ; therefore, this version of the traveling salesman problem lies in the class NP. Similarly, through the device of introducing a target number T , all the combinatorial optimization problems normally considered in the fields of commerce, science, and engineering have versions that lie in the class NP.

So NP is the area into which combinatorial problems typically fall; within NP lies P, the class of problems that have efficient solutions. A fundamental question is, What is the relationship between the class P and the class NP? It is clear that P is a subset of NP, and the question that Cook drew attention to is whether P and NP might be the same class. If P were equal to NP, there would be astounding consequences: It would mean that every problem for which solutions are easy to check would also be easy to solve; it would mean that, whenever a theorem had a short proof, a uniform procedure would be able to find that proof quickly; it would mean that all the usual combinatorial optimization problems would be solvable in polynomial time. In short, it would mean that the curse of combinatorial explosions could be eradicated. But, despite all this heuristic evidence that it would be too good to be true if P and NP were equal, no proof that $P \neq NP$ has ever been found, and some experts even believe that no proof will ever be found.

The most important achievement of Cook's paper was to show that $P = NP$ if and only if a particular computational problem called the Satisfiability Problem lies in P. The Satisfiability Problem comes from mathematical logic and has applications in switching theory, but it can be stated as a simple combinatorial puzzle: Given several sequences of upper- and lowercase letters, is it possible to select a letter from each sequence without selecting both the upper- and lowercase versions of any letter? For example, if the sequences are Abc , BC , aB , and ac it is possible to choose A from the first sequence, B from the second and third, and c from the fourth; note that the same letter can be chosen more than once, provided we do not choose both its uppercase and lowercase versions. An example where there is no way to make the required selections is given by the four sequences AB , Ab , aB , and ab .

The Satisfiability Problem is clearly in NP, since it is easy to check whether a proposed selection of letters satisfies the conditions of the problem. Cook proved that, if the Satisfiability Problem is solvable in polynomial time, then every problem in NP is solvable in polynomial time, so that $P = NP$. Thus we see that this seemingly bizarre and inconsequential problem is an archetypal combinatorial problem; it holds the key to the efficient solution of all problems in NP.

Cook's proof was based on the concept of reducibility that we encountered earlier in our discussion of computability theory. He showed that any instance of a problem in NP can be transformed into a corresponding instance of the Satisfiability Problem in such a way that the original has a solution if and only if the satisfiability instance does. Moreover, this translation can be accomplished in polynomial time. In other words, the Satisfiability Problem is general enough to capture the structure of any problem in NP. It follows that, if we could solve the Satisfiability Problem in polynomial time, then we would be able to construct a polynomial-time algorithm to solve any problem in NP. This algorithm would consist of two parts: a polynomial-time translation procedure that converts instances of the given problem into instances of the Satisfiability Problem, and a polynomial-time subroutine to solve the Satisfiability Problem itself (see Figure 6).

Upon reading Cook's paper, I realized at once that his concept of an archetypal combinatorial problem was a formalization of an idea that had long been part of the folklore of combinatorial optimization. Workers in that field knew that the integer programming problem, which is essentially the problem of deciding whether a system of linear inequalities has a solution in integers, was general enough to express the constraints of any of the commonly encountered combinatorial optimization problems. Dantzig had published a paper on that theme in 1960. Because Cook was interested in theorem proving rather than combinatorial optimization, he had chosen a different archetypal problem, but the basic idea was the same. However, there was a key difference: By using the apparatus of complexity theory, Cook had created a framework within which the archetypal nature of a given problem could become a theorem, rather than an informal thesis. Interestingly, Leonid Levin, who was then in Leningrad and is now a professor at Boston University, independently discovered essentially the same set of ideas. His archetypal problem had to do with tilings of finite regions of the plane with dominoes.

I decided to investigate whether certain classic combinatorial problems, long believed to be intractable,

were also archetypal in Cook's sense. I called such problems "polynomial complete," but that term became superseded by the more precise term "NP-complete." A problem is NP-complete if it lies in the class NP, and every problem in NP is polynomial-time reducible to it. Thus, by Cook's theorem, the Satisfiability Problem is NP-complete. To prove that a given problem in NP is NP-complete, it suffices to show that some problem already known to be NP-complete is polynomial-time reducible to the given problem. By constructing a series of polynomial-time reductions, I showed that most of the classical problems of packing, covering, matching, partitioning, routing, and scheduling that arise in combinatorial optimization are NP-complete. I presented these results in 1972 in a paper called "Reducibility among Combinatorial Problems." My early results were quickly refined and extended by other workers, and in the next few years, hundreds of different problems, arising in virtually every field where computation is done, were shown to be NP-complete.

COPING WITH NP-COMPLETE PROBLEMS

I was rewarded for my research on NP-complete problems with an administrative post. From 1973 to 1975, I headed the newly formed Computer Science Division at Berkeley, and my duties left me little time for research. As a result, I sat on the sidelines during a very active period, during which many examples of NP-complete problems were found, and the first attempts to get around the negative implications of NP-completeness got under way.

The NP-completeness results proved in the early 1970s showed that, unless $P = NP$, the great majority of the problems of combinatorial optimization that arise in commerce, science, and engineering are intractable: No methods for their solution can completely evade combinatorial explosions. How, then, are we to cope with such problems in practice? One possible approach stems from the fact that near-optimal solutions will often be good enough: A traveling salesman will probably be satisfied with a tour that is a few percent longer than the optimal one. Pursuing this approach, researchers began to search for polynomial-time algorithms that were guaranteed to produce near-optimal solutions to NP-complete combinatorial optimization problems. In most cases, the performance guarantee for the approximation algorithm was in the form of an upper bound on the ratio between the cost of the solution produced by the algorithm and the cost of an optimal solution.

Some of the most interesting work on approximation algorithms with performance guarantees concerned the one-dimensional bin-packing problem. In this problem, a collection of items of various sizes must be packed into bins, all of which have the same capacity. The goal is to minimize the number of bins used for the packing, subject to the constraint that the sum of the sizes of the items packed into any bin may not exceed the bin capacity. During the mid 1970s, a series of papers on approximation algorithms for bin packing culminated in David Johnson's analysis of the first-fit-decreasing algo-

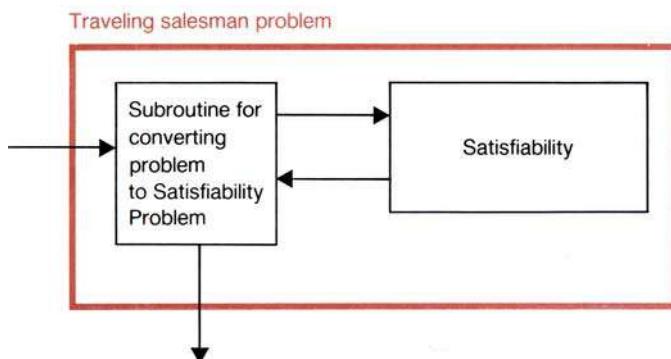


FIGURE 6. The Traveling Salesman Problem Is Polynomial-Time Reducible to the Satisfiability Problem

rithm. In this simple algorithm, the items are considered in decreasing order of their sizes, and each item in turn is placed in the first bin that can accept it. In the example in Figure 7, for instance, there are four bins each with a capacity of 10, and eight items ranging in size from 2 to 8. Johnson showed that this simple method was guaranteed to achieve a relative error of at most $2/9$; in other words, the number of bins required was never more than about 22 percent greater than the number of bins in an optimal solution. Several years later, these results were improved still further, and it was eventually shown that the relative error could be made as small as one liked, although the polynomial-time algorithms required for this purpose lacked the simplicity of the first-fit-decreasing algorithm that Johnson analyzed.

The research on polynomial-time approximation algorithms revealed interesting distinctions among the NP-complete combinatorial optimization problems. For some problems, the relative error can be made as small as one likes; for others, it can be brought down to a certain level, but seemingly no further; other problems have resisted all attempts to find an algorithm with bounded relative error; and finally, there are certain problems for which the existence of a polynomial-time approximation algorithm with bounded relative error would imply that $P = NP$.

During the sabbatical year that followed my term as an administrator, I began to reflect on the gap between theory and practice in the field of combinatorial optimization. On the theoretical side, the news was bleak. Nearly all the problems one wanted to solve were NP-complete, and in most cases, polynomial-time approximation algorithms could not provide the kinds of performance guarantees that would be useful in practice. Nevertheless, there were many algorithms that seemed to work perfectly well in practice, even though they lacked a theoretical pedigree. For example, Lin and Kernighan had developed a very successful local improvement strategy for the traveling salesman problem. Their algorithm simply started with a random tour and kept improving it by adding and deleting a few links, until a tour was eventually created that could not be improved by such local changes. On contrived in-

stances, their algorithm performed disastrously, but in practical instances, it could be relied on to give nearly optimal solutions. A similar situation prevailed for the simplex algorithm, one of the most important of all computational methods: It reliably solved the large linear programming problems that arose in applications, despite the fact that certain artificially constructed examples caused it to run for an exponential number of steps.

It seemed that the success of such inexact or rule-of-thumb algorithms was an empirical phenomenon that needed to be explained. And it further seemed that the explanation of this phenomenon would inevitably require a departure from the traditional paradigms of complexity theory, which evaluate an algorithm according to its performance on the worst possible input that can be presented to it. The traditional worst-case analysis—the dominant strain in complexity theory—corresponds to a scenario in which the instances of a problem to be solved are constructed by an infinitely intelligent adversary who knows the structure of the algorithm and chooses inputs that will embarrass it to the maximal extent. This scenario leads to the conclusion that the simplex algorithm and the Lin-Kernighan algorithm are hopelessly defective. I began to pursue another approach, in which the inputs are assumed to come from a user who simply draws his instances from some reasonable probability distribution, attempting neither to foil nor to help the algorithm.

In 1975 I decided to bite the bullet and commit myself to an investigation of the probabilistic analysis of combinatorial algorithms. I must say that this decision required some courage, since this line of research had its detractors, who pointed out quite correctly that there was no way to know what inputs were going to be presented to an algorithm, and that the best kind of guarantees, if one could get them, would be worst-case guarantees. I felt, however, that in the case of NP-complete problems we weren't going to get the worst-case guarantees we wanted, and that the probabilistic approach was the best way and perhaps the only way to understand why heuristic combinatorial algorithms worked so well in practice.

Probabilistic analysis starts from the assumption that the instances of a problem are drawn from a specified probability distribution. In the case of the traveling salesman problem, for example, one possible assumption is that the locations of the n cities are drawn independently from the uniform distribution over the unit square. Subject to this assumption, we can study the probability distribution of the length of the optimal tour or the length of the tour produced by a particular algorithm. Ideally, the goal is to prove that some simple algorithm produces optimal or near-optimal solutions with high probability. Of course, such a result is meaningful only if the assumed probability distribution of problem instances bears some resemblance to the population of instances that arise in real life, or if the probabilistic analysis is robust enough to be valid for a wide range of probability distributions.

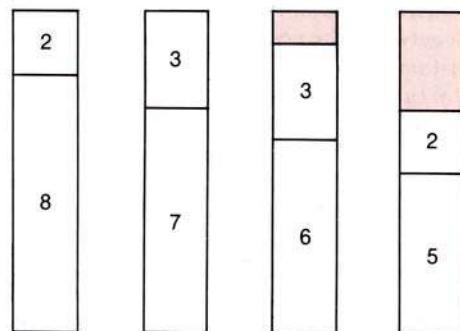


FIGURE 7. A Packing Created by the First-Fit Decreasing Algorithm

Among the most striking phenomena of probability theory are the laws of large numbers, which tell us that the cumulative effect of a large number of random events is highly predictable, even though the outcomes of the individual events are highly unpredictable. For example, we can confidently predict that, in a long series of flips of a fair coin, about half the outcomes will be heads. Probabilistic analysis has revealed that the same phenomenon governs the behavior of many combinatorial optimization algorithms when the input data are drawn from a simple probability distribution: With very high probability, the execution of the algorithm evolves in a highly predictable fashion, and the solution produced is nearly optimal. For example, a

1960 paper by Beardwood, Halton, and Hammersley shows that, if the n cities in a traveling salesman problem are drawn independently from the uniform distribution over the unit square, then, when n is very large, the length of the optimal tour will almost surely be very close to a certain absolute constant times the square root of the number of cities. Motivated by their result, I showed that, when the number of cities is extremely large, a simple divide-and-conquer algorithm will almost surely produce a tour whose length is very close to the length of an optimal tour (see Figure 8). The algorithm starts by partitioning the region where the cities lie into rectangles, each of which contains a small number of cities. It then constructs an optimal

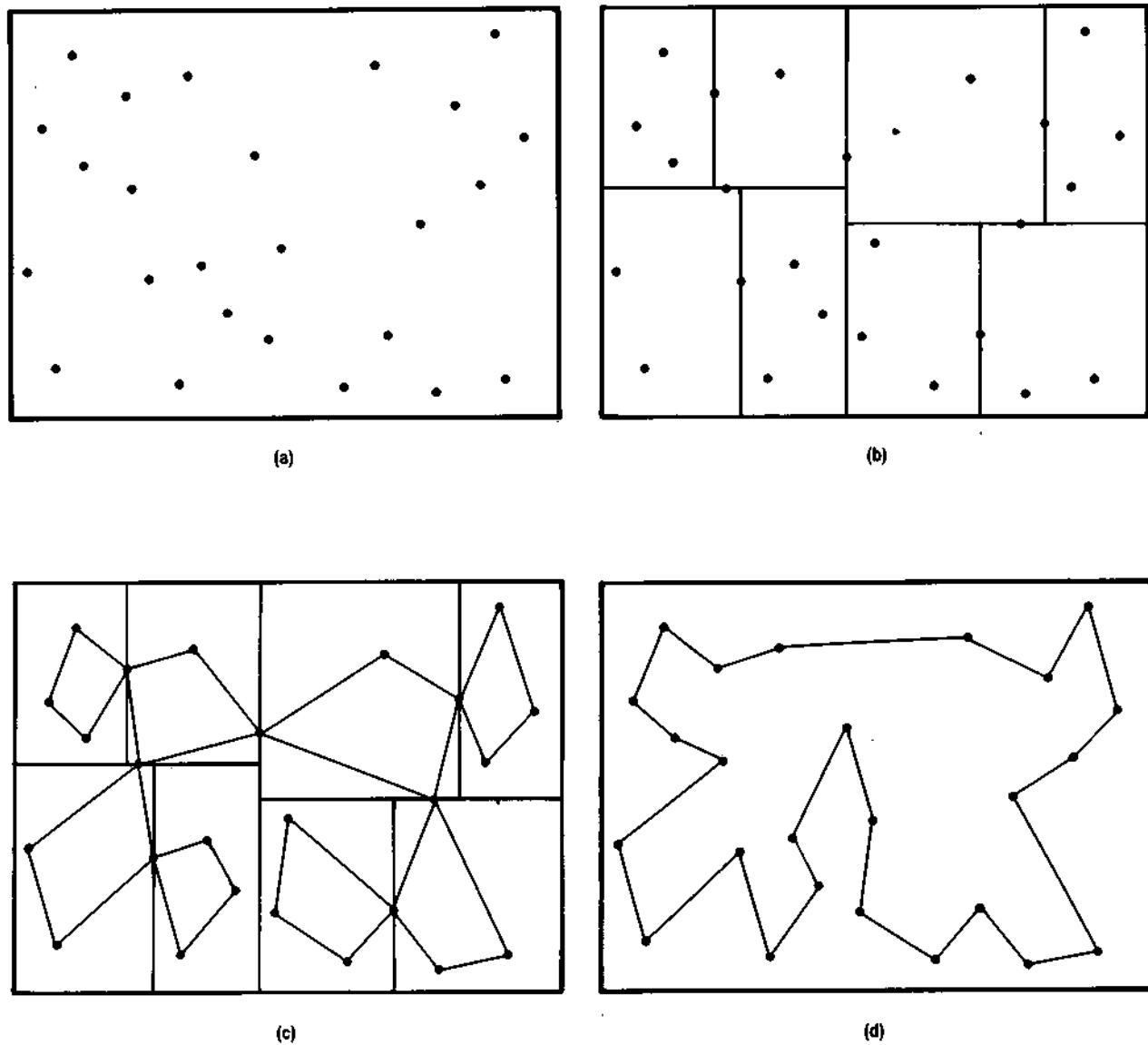


FIGURE 8. A Divide-and-Conquer Algorithm for the Traveling Salesman Problem in the Plane

tour through the cities in each rectangle. The union of all these little tours closely resembles an overall traveling salesman tour, but differs from it because of extra visits to those cities that lie on the boundaries of the rectangles. Finally, the algorithm performs a kind of local surgery to eliminate these redundant visits and produce a tour.

Many further examples can be cited in which simple approximation algorithms almost surely give near-optimal solutions to random large instances of NP-complete optimization problems. For example, my student Sally Floyd, building on earlier work on bin packing by Bentley, Johnson, Leighton, McGeoch, and McGeoch, recently showed that, if the items to be packed are drawn independently from the uniform distribution over the interval $(0, 1/2]$, then, no matter how many items there are, the first-fit decreasing algorithm will almost surely produce a packing with less than 10 bins worth of wasted space.

Some of the most notable applications of probabilistic analysis have been to the linear programming problem. Geometrically, this problem amounts to finding the vertex of a polyhedron closest to some external hyperplane. Algebraically, it is equivalent to minimizing a linear function subject to linear inequality constraints. The linear function measures the distance to the hyperplane, and the linear inequality constraints correspond to the hyperplanes that bound the polyhedron.

The simplex algorithm for the linear programming problem is a hill-climbing method. It repeatedly slides from vertex to neighboring vertex, always moving closer to the external hyperplane. The algorithm terminates when it reaches a vertex closer to this hyperplane than any neighboring vertex; such a vertex is guaranteed to be an optimal solution. In the worst case, the simplex algorithm requires a number of iterations that grow exponentially with the number of linear inequalities needed to describe the polyhedron, but in practice, the number of iterations is seldom greater than three or four times the number of linear inequalities.

Karl-Heinz Borgwardt of West Germany and Steve Smale of Berkeley were the first researchers to use probabilistic analysis to explain the unreasonable success of the simplex algorithm and its variants. Their analyses hinged on the evaluation of certain multidimensional integrals. With my limited background in mathematical analysis, I found their methods impenetrable. Fortunately, one of my colleagues at Berkeley, Ilan Adler, suggested an approach that promised to lead to a probabilistic analysis in which there would be virtually no calculation; one would use certain symmetry principles to do the required averaging and magically come up with the answer.

Pursuing this line of research, Adler, Ron Shamir, and I showed in 1983 that, under a reasonably wide range of probabilistic assumptions, the expected number of iterations executed by a certain version of the simplex algorithm grows only as the square of the number of linear inequalities. The same result was also

obtained via multidimensional integrals by Michael Todd and by Adler and Nimrod Megiddo. I believe that these results contribute significantly to our understanding of why the simplex method performs so well.

The probabilistic analysis of combinatorial optimization algorithms has been a major theme in my research over the past decade. In 1975, when I first committed myself to this research direction, there were very few examples of this type of analysis. By now there are hundreds of papers on the subject, and all of the classic combinatorial optimization problems have been subjected to probabilistic analysis. The results have provided a considerable understanding of the extent to which these problems can be tamed in practice. Nevertheless, I consider the venture to be only partially successful. Because of the limitations of our techniques, we continue to work with the most simplistic of probabilistic models, and even then, many of the most interesting and successful algorithms are beyond the scope of our analysis. When all is said and done, the design of practical combinatorial optimization algorithms remains as much an art as it is a science.

RANDOMIZED ALGORITHMS

Algorithms that toss coins in the course of their execution have been proposed from time to time since the earliest days of computers, but the systematic study of such randomized algorithms only began around 1976. Interest in the subject was sparked by two surprisingly efficient randomized algorithms for testing whether a number n is prime; one of these algorithms was proposed by Solovay and Volker Strassen, and the other by Rabin. A subsequent paper by Rabin gave further examples and motivation for the systematic study of randomized algorithms, and the doctoral thesis of John Gill, under the direction of my colleague Blum, laid the foundations for a general theory of randomized algorithms.

To understand the advantages of coin tossing, let us turn again to the scenario associated with worst-case analysis, in which an all-knowing adversary selects the instances that will tax a given algorithm most severely. Randomization makes the behavior of an algorithm unpredictable even when the instance is fixed, and thus can make it difficult, or even impossible, for the adversary to select an instance that is likely to cause trouble. There is a useful analogy with football, in which the algorithm corresponds to the offensive team and the adversary to the defense. A deterministic algorithm is like a team that is completely predictable in its play calling, permitting the other team to stack its defenses. As any quarterback knows, a little diversification in the play calling is essential for keeping the defensive team honest.

As a concrete illustration of the advantages of coin tossing, I present a simple randomized pattern-matching algorithm invented by Rabin and myself in 1980. The pattern-matching problem is a fundamental one in text processing. Given a string of n bits called

Pattern	11001
Text	011011101 1100100

FIGURE 9. A Pattern-Matching Problem

the pattern, and a much longer bit string called the text, the problem is to determine whether the pattern occurs as a consecutive block within the text (see Figure 9). A brute-force method of solving this problem is to compare the pattern directly with every n -bit block within the text. In the worst case, the execution time of this method is proportional to the product of the length of the pattern and the length of the text. In many text processing applications, this method is unacceptably slow unless the pattern is very short.

Our method gets around the difficulty by a simple hashing trick. We define a "fingerprinting function" that associates with each string of n bits a much shorter string called its *fingerprint*. The fingerprinting function is chosen so that it is possible to march through the text, rapidly computing the fingerprint of every n -bit-long block. Then, instead of comparing the pattern with each such block of text, we compare the fingerprint of the pattern with the fingerprint of every such block. If the fingerprint of the pattern differs from the fingerprint of each block, then we know that the pattern does not occur as a block within the text.

The method of comparing short fingerprints instead of long strings greatly reduces the running time, but it leads to the possibility of false matches, which occur when some block of text has the same fingerprint as the pattern, even though the pattern and the block of text are unequal. False matches are a serious problem; in fact, for any particular choice of fingerprinting function it is possible for an adversary to construct an example of a pattern and a text such that a false match occurs at every position of the text. Thus, some backup method is needed to defend against false matches, and the advantages of the fingerprinting method seem to be lost.

Fortunately, the advantages of fingerprinting can be restored through randomization. Instead of working with a single fingerprinting function, the randomized method has at its disposal a large family of different easy-to-compute fingerprinting functions. Whenever a problem instance, consisting of a pattern and a text, is presented, the algorithm selects a fingerprinting function at random from this large family, and uses that function to test for matches between the pattern and the text. Because the fingerprinting function is not known in advance, it is impossible for an adversary to construct a problem instance that is likely to lead to false matches; it can be shown that, no matter how the pattern and the text are selected, the probability of a false match is very small. For example, if the pattern is 250 bits long and the text is 4000 bits long, one can

work with easy-to-compute 32-bit fingerprints and still guarantee that the probability of a false match is less than one in a thousand in every possible instance. In many text processing applications, this probabilistic guarantee is good enough to eliminate the need for a backup routine, and thus the advantages of the fingerprinting approach are regained.

Randomized algorithms and probabilistic analysis of algorithms are two contrasting ways to depart from the worst-case analysis of deterministic algorithms. In the former case, randomness is injected into the behavior of the algorithm itself, and in the latter case, randomness is assumed to be present in the choice of problem instances. The approach based on randomized algorithms is, of course, the more appealing of the two, since it avoids assumptions about the environment in which the algorithm will be used. However, randomized algorithms have not yet proved effective in combating the combinatorial explosions characteristic of NP-complete problems, and so it appears that both of these approaches will continue to have their uses.

CONCLUSION

This brings me to the end of my story, and I would like to conclude with a brief remark about what it's like to be working in theoretical computer science today. Whenever I participate in the annual ACM Theory of Computing Symposium, or attend the monthly Bay Area Theory Seminar, or go up the hill behind the Berkeley campus to the Mathematical Sciences Research Institute, where a year-long program in computational complexity is taking place, I am struck by the caliber of the work that is being done in this field. I am proud to be associated with a field of research in which so much excellent work is being done, and pleased that I'm in a position, from time to time, to help prodigiously talented young researchers get their bearings in this field. Thank you for giving me the opportunity to serve as a representative of my field on this occasion.

CR Categories and Subject Descriptors: A.0 [General Literature]: biographies/autobiographies; F.0 [Theory of Computation]: General; F.1.1 [Computation by Abstract Devices]: Models of Computation—*computability theory*; F.1.2 [Computation by Abstract Devices]: Modes of Computation—*parallelism, probabilistic computation*; F.1.3 [Computation by Abstract Devices]: Complexity Classes—*reducibility and completeness, relations among complexity classes*; F.2.0 [Analysis of Algorithms and Problem Complexity]: General; G.2.1 [Discrete Mathematics]: Combinatorics; G.2.2 [Discrete Mathematics]: Graph Theory; K.2 [History of Computing]: people

General Terms: Performance, Theory

Additional Key Words and Phrases: Richard Karp, Turing Award

Author's Present Address: Richard Karp, 571 Evans Hall, University of California, Berkeley, CA 94720.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1970 Turing Lecture

Some Comments from a Numerical Analyst

J. H. WILKINSON

National Physical Laboratory, Teddington, Middlesex, England

ABSTRACT. A description is given of life with A.M. Turing at the National Physical Laboratory in the early days of the development of electronic computers (1946-1948). The present mood of pessimism among numerical analysts resulting from difficult relationships with computer scientists and mathematicians is discussed. It is suggested that in light of past and present performance this pessimism is unjustified and is the main enemy of progress in numerical mathematics. Some achievements in the fields of matrix computations and error analysis are discussed and likely changes in the direction of research in numerical analysis are sketched.

KEY WORDS AND PHRASES: Turing, National Physical Laboratory, ACE and PILOT ACE, Woodger, Huskey, matrix computations, Faddeeva, error analysis, Turing's contribution, mathematical physics, applied mathematics

CR CATEGORIES: 1.2, 5.10, 5.11, 5.14

Introduction

When at last I recovered from the feeling of shocked elation at being invited to give the 1970 Turing Award Lecture, I became aware that I must indeed prepare an appropriate lecture. There appears to be a tradition that a Turing Lecturer should decide for himself what is expected from him, and probably for this reason previous lectures have differed considerably in style and content. However, it was made quite clear that I was to give an after-luncheon speech and that I would not have the benefit of an overhead projector or a blackboard.

Although I have been associated with high speed computers since the pioneering days, my main claim, such as it is, to the honor of giving the 1970 lecture rests on my work as a numerical analyst, particularly in the field of error analysis. A study of the program for this meeting revealed that numerical analysis was conspicuous by its absence, and accordingly I felt that it would be inappropriate to prepare a rather heavy discourse on rounding errors; indeed I doubt whether it would be a suitable topic for an after-luncheon speech in any setting. I decided therefore to make some rather personal comments based on my experience as a numerical analyst over the last twenty-five years.

There is one important respect in which it is reasonably probable that I shall occupy a unique position among Turing Lecturers.

Maurice Wilkes, giving the 1967 Turing Lecture remarked that it was unlikely that many of those who followed him would be people who were acquainted with Alan Turing. In fact I can claim a good deal more than that. From 1946 to 1948 I had the privilege of working with the great man himself at the National Physical Laboratory. I use the term "great man" advisedly because he was indeed a remarkable genius. To those of us at N.P.L. who knew him and worked with him it has

been a source of great pleasure that the ACM should have recognized his outstanding contributions to computer science by founding this Turing Award, and because of my connection with his work at an important period in his career, it is particularly gratifying for me to be a recipient. I trust that in the circumstances it will not be regarded as inappropriate if I devote a part of my lecture to the period I spent working with him. My career was certainly profoundly influenced by the association and, without it, it is unlikely that I would have remained in the computer field.

Life with Alan Turing

I was by inclination and training a classical analyst. Cambridge was still dominated by classical analysis in the '30s and I was strongly influenced by the Hardy-Littlewood tradition. Had it not been for World War II, I would almost certainly have taken my Ph.D. in that field. However, I was of military age when the war broke out and being by nature a patriotic man I felt that I could serve my country more effectively, and incidentally a lot more comfortably, working in the Government Scientific Service, than serving as an incompetent infantryman. The British Government took a surprisingly enlightened attitude on the subject, and almost from the start of the war those with scientific qualifications were encouraged to take this course.

I therefore spent the war in the Armament Research Department, which has much in common with Aberdeen Proving Ground, working mainly on such fascinating topics as external ballistics, fragmentation of bombs and shells, and the thermodynamics of explosives. My task was to solve problems of a mathematical nature arising in these fields, using computational methods if necessary. Anybody who has ever been subjected to this discipline will know that it can be quite a chastening experience and successes are heavily diluted with failures. I did not at first find this task particularly congenial, but gradually I became interested in the numerical solution of physical problems. Later in this lecture I shall describe an early experience with matrix computations which was to have a considerable influence on my subsequent career.

It was not possible to obtain an immediate release at the end of the war and in 1946 I joined the newly formed Mathematics Division at the National Physical Laboratory. It was there that I first met Alan Turing, though he was, of course, known to me before by reputation, but mainly as an eccentric. It is interesting to recall now that computer science virtually embraces two whole divisions at N.P.L. and spreads its tentacles into the remainder, that at that time the staff of the high speed computing section (or ACE section as it was called) numbered $1\frac{1}{2}$. The one, of course, was no less a person than Alan Turing himself and I was the half. I hasten to add that this doesn't represent false modesty on my part. I was to spend half my time in the computing section, which was in the capable hands of Charles Goodwin and Leslie Fox, and the other half with Alan Turing. For several months Alan and I worked together in a remarkably small room at the top of an old house which had been taken over "temporarily" by N.P.L. to house Mathematics Division. Needless to say, twenty-five years later it is still part of N.P.L. Turing never became an empire builder; he assembled his staff rather slowly and worked rather intimately with them. A year later the staff had reached only $3\frac{1}{2}$, the two additions being Mike

Woodger, who is best known for his work on Algol, and Harry Huskey (who needs no introduction to ACM audiences) who spent 1947 at N.P.L.

My task was to assist Turing in the logical design of the computer ACE which was to be built at N.P.L. and to consider the problems of programming some of the more basic algorithms of numerical analysis, and my work in the Computing Section was intended to broaden my knowledge of that subject. (Those of you who are familiar with Turing's work will be interested to know that he referred to the sets of instructions needed for a particular problem as the relevant "instruction table," a term which later led to misunderstandings with people elsewhere.) As you can imagine, this left me with little idle time. Working with Turing was tremendously stimulating, perhaps at times to the point of exhaustion. He had recently become keenly interested in the problems of numerical analysis himself, and he took great pleasure in subjecting Leslie Fox, who was our most experienced numerical analyst at N.P.L., to penetrating but helpful criticisms of the methods he was using.

It was impossible to work "half-time" for a man like Turing and almost from the start the periods spent with the computing section were rather brief. The joint appointment did, however, have its useful aspect. Turing occasionally had days when he was "unapproachable" and at such times it was advisable to exercise discretion. I soon learned to recognize the symptoms and would exercise my right (or, as I usually put it, "meet my obligations") of working in the Computing Section until the mood passed, which it usually did quite quickly.

Turing had a strong predilection for working things out from first principles, usually in the first instance without consulting any previous work on the subject, and no doubt it was this habit which gave his work that characteristically original flavor. I was reminded of a remark which Beethoven is reputed to have made when he was asked if he had heard a certain work of Mozart which was attracting much attention. He replied that he had not, and added "neither shall I do so, lest I forfeit some of my own originality."

Turing carried this to extreme lengths and I must confess that at first I found it rather irritating. He would set me a piece of work and when I had completed it he would not deign to look at my solution but would embark on the problem himself; only after having a preliminary trial on his own was he prepared to read my work. I soon came to see the advantage of his approach. In the first place he was really not as quick at grasping other people's ideas as he was at formulating his own, but what is more important, he would frequently come up with some original approach which had escaped me and might well have eluded him, had he read my account immediately. When he finally got round to reading my own work he was generally very appreciative; he was particularly fond of little programming tricks (some people would say that he was too fond of them to be a "good" programmer) and would chuckle with boyish good humor at any little tricks I may have used.

When I joined N.P.L. I had not made up my mind to stay permanently and still thought in terms of returning to Cambridge to take up research in classical analysis. The period with Turing fired me with so much enthusiasm for the computer project and so heightened my interest in numerical analysis that gradually I abandoned this idea. As I rather like to put it when speaking to pure mathematical friends, "had it not been for Turing I would probably have become just a pure mathematician," taking care to give the remark a suitably pejorative flavor.

Turing's reputation is now so well established that it scarcely stands in need of a

boost from me. However, I feel bound to say that his published work fails to give an adequate impression of his remarkable versatility as a mathematician. His knowledge ranged widely over the whole field of pure and applied mathematics and seemed, as it were, not merely something he had learned from books, but to form an integral part of the man himself. One could scarcely imagine that he would ever "forget" any of it. In spite of this he had only twenty published papers to his credit (and this only if one includes virtually everything), written over a period of some twenty years. Remarkable as some of these papers are, this work represents a mere fraction of what he might have done if things had turned out just a little differently.

In the first place there were the six years starting from 1939 which he spent at the Foreign Office. He was 27 in 1939, so that in different circumstances this period might well have been the most productive of his life. He seemed not to have regretted the years he spent there and indeed we formed the impression that this was one of the happiest times of his life. Turing simply loved problems and puzzles of all kinds and the problems he encountered there must have given him a good deal of fun. Certainly it was there that he gained his knowledge of electronics and this was probably the decisive factor in his deciding to go to N.P.L. to design an electronic computer rather than returning to Cambridge. Mathematicians are inclined to refer to this period as the "wasted years" but I think he was too broad a scientist to think of it in such terms.

A second factor limiting his output was a marked disinclination to put pen to paper. At school he is reputed to have had little enthusiasm for the "English subjects" and he seemed to find the tedium of publishing a paper even more oppressive than most of us do. For myself I find his style of writing rather refreshing and full of little personal touches which are particularly attractive to someone who knew him. When in the throes of composition he would hammer away on an old typewriter (he was an indifferent typist, to put it charitably) and it was on such occasions that visits to the Computing Section were particularly advisable.

While I was preparing this talk an early Mathematics Division report was unearthed. It was written by Turing in 1946 for the Executive Committee of N.P.L., and its main purpose was to convince the committee of the feasibility and importance of building an electronic computer. It is full of characteristic touches of humor, and rereading it for the first time for perhaps 24 years I was struck once again by his remarkable originality and versatility. It is perhaps salutary to be reminded that as early as 1946 Turing had considered the possibility of working with both interval and significant digit arithmetic and the report recalled forgotten conversations, not to mention heated arguments, which we had on this topic.

Turing's international reputation rests mainly on his work on computable numbers but I like to recall that he was a considerable numerical analyst, and a good part of his time from 1946 onwards was spent working in this field, though mainly in connection with the solution of physical problems. While at N.P.L. he wrote a remarkable paper on the error analysis of matrix computations [1] and I shall return to this later.

During the last few months at N.P.L., Turing became increasingly dissatisfied with progress on the ACE project. He had always thought in terms of a large machine with 200 long delay lines storing some 6,000 words and I think this was too ambitious a project for the resources of N.P.L. (and indeed of most other places) at that time.

During his visit Harry Huskey attempted to get work started on a less ambitious machine, based on Turing's ideas. Alan could never bring himself to support this project and in 1948 he left N.P.L. to join the group at Manchester University. After he left, the four senior members of the ACE section of Mathematics Division and the recently formed Electronics Section joined forces and collaborated on the construction of the computer PILOT ACE, for which we took over some of the ideas we had worked out with Harry Huskey; for the next two to three years we all worked as electronic engineers. I think we can claim that the PILOT ACE was a complete success and since Turing would not have permitted this project to get off the ground, to this extent at least we benefitted from his departure, though the Mathematics Division was never quite the same again. Working with a genius has both advantages and disadvantages! Once the machine was a success, however, there were no sour grapes from Turing and he was always extremely generous about what had been achieved.

The Present State of Numerical Analysis

I would now like to come to the main theme of my lecture, the present status of numerical analysis. Numerical analysis is unique among the various topics which comprise the rather ill-defined discipline of computer science. I make this remark rather defiantly because I would be very sorry to see numerical analysis sever all its connections with computer science, though I recognize that my views must be influenced to some extent by having worked in the exciting pioneering days on the construction of electronic computers. Nevertheless, numerical analysis is clearly different from the other topics in having had a long and distinguished history. Only the name is new (it appears not to have been used before the '50s) and this at least it has in common with computer science.

Some like to trace its history back to the Babylonians and if one chooses to regard any reasonably systematic computation as numerical analysis I suppose this is justifiable. Certainly many of the giants of the mathematical world, including both the great Newton and Gauss themselves, devoted a substantial part of their research to computational problems. In those days it was possible for a mathematician to spend his time in this way without being apprehensive of the criticism of his colleagues.

Many of the leaders of the computer revolution thought in terms of developing a tool which was specifically intended for the solution of problems arising in physics and engineering. This was certainly true of the two men of genius, von Neumann and Turing, who did so much to attract people of real ability into the computing field in the early days. The report of Turing to which I referred earlier makes it quite clear that he regarded such applications as the main justification for embarking on what was, even then, a comparatively expensive undertaking. A high percentage of the leading lights of the newly formed computer societies were primarily numerical analysts and the editorial boards of the new journals were drawn largely from their ranks.

The use of electronic computers brought with it a new crop of problems all perhaps loosely associated with "programming" and quite soon a whole field of new endeavors grew up around the computer. In a brilliant article on numerical analysis [2] Philip Davis uses the term "computerology" to encompass these multifarious

activities but is careful to attribute the term to an unnamed friendly critic. I do not intend to use the term in a pejorative sense in this talk, but it is a useful collective word to cover everything in computer science other than numerical analysis. Many people who set out originally to solve some problem in mathematical physics found themselves temporarily deflected by the problems of computerology and we are still waiting with bated breath for the epoch-making contributions they will surely make when they return to the fold, clothed in their superior wisdom.

In contrast to numerical analysis the problems of computerology are entirely new. The whole science is characterized by restless activity and excitement and completely new topics are constantly springing up. Although, no doubt, a number of the new activities will prove to be short-lived, computerology has a vital part to play in ensuring that computers are fully exploited. I'm sure that it is good for numerical analysts to be associated with a group of people who are so much alive and full of enthusiasm. I'm equally sure that there is merit in computer science embracing a subject like numerical analysis which has a solid background of past achievement. Inevitably though, numerical analysis has begun to look a little square in the computer science setting, and numerical analysts are beginning to show signs of losing faith in themselves. Their sense of isolation is accentuated by the present trend towards abstraction in mathematics departments which makes for an uneasy relationship. How different things might have been if the computer revolution had taken place in the 19th century! In his article Davis remarks that people are already beginning to ask, "Is numerical analysis dead?" Davis has given his own answer to this question and I do not propose to pursue it here. In any case "numerical analysts" may be likened to "The Establishment" in computer science and in all spheres it is fashionable to diagnose "rigor mortis" in the Establishment.

There is a second question which is asked with increasing frequency. It assumes many different guises but is perhaps best expressed by the catch-phrase, "What's new in numerical analysis?" This is invariably delivered in such a manner as to leave no doubt that the questioner's answer is "Nothing," or, more probably, one of the more vulgar two-word synonyms, in which the English language is so rich. This criticism reminds me of a somewhat similar situation which exists with respect to functional analysis. Those brought up in an older tradition are inclined to say that "there is nothing new in functional analysis, it merely represents a dressing up of old results in new clothes." There is just sufficient truth in this to confirm the critics in their folly.

In my opinion the implied criticism involves a false comparison. Of course everything in computerology is new; that is at once its attraction, and its weakness. Only recently I learned that computers are revolutionizing astrology. Horoscopes by computer!—it's certainly never been done before, and I understand that it is very remunerative! Seriously though, it was not to be expected that numerical analysis would be turned upside down in the course of a decade or two, just because we had given it a new name and at last had satisfactory tools to work with. Over the last 300 years some of the finest intellects in the mathematical world have been brought to bear on the problems we are trying to solve. It is not surprising that our rate of progress cannot quite match the heady pace which is so characteristic of computerology.

Some Achievements in Numerical Analysis

In case you are tempted to think that I am about to embark on excuses for not having made any real progress, I hasten to assure you that I have no such intention. While I was preparing this lecture I made a brief review of what has been achieved since 1950 and found it surprisingly impressive. In the next few minutes I would like to say just a little about the achievements in the area with which I am best acquainted, matrix computations.

We are fortunate here in having, in the little book written by V. N. Faddeeva [3], an admirably concise and accurate account of the situation as it was in 1950. A substantial part of the book is devoted to the solution of the eigenvalue problem, and scarcely any of the methods discussed there are in use today. In fact as far as non-Hermitian matrices are concerned, even the methods which were advocated at the 1957 Wayne matrix conference have been almost completely superseded. Using a modern version of the QR algorithm one can expect to produce an accurate eigen-system of a dense matrix of order 100 in a time which is of the order of a minute. One can then go on to produce rigorous error bounds for both the eigenvalues and eigenvectors if required, deriving a more accurate system as a byproduct. At the 1957 Wayne conference we did not appear to be within hailing distance of such an achievement. A particularly pleasing feature of the progress is that it is an appreciation of the problem of numerical stability resulting from advances in error analysis that has played a valuable part in suggesting the new algorithms.

Comparable advances have been made in the development of iterative methods for solving sparse linear systems of the type arising from partial differential equations; here algorithmic advances have proceeded *pari passu* with a deepening understanding of the convergence properties of iterative methods. As far as dense systems are concerned the development of new algorithms has been less important, but our understanding of the stability of the standard methods has undergone a complete transformation.

In this connection I would like to make my last remarks about life with Turing. When I joined N.P.L. in 1946 the mood of pessimism about the stability of elimination methods for solving linear systems was at its height and was a major talking point. Bounds had been produced which purported to show that the error in the solution would be proportional to 4^n and this suggested that it would be impractical to solve systems even of quite modest order. I think it was true to say that at that time (1946) it was the more distinguished mathematicians who were most pessimistic, the less gifted being perhaps unable to appreciate the full severity of the difficulties. I do not intend to indicate my place on this scale, but I did find myself in a rather uncomfortable position for the following reason.

It so happens that while I was at the Armament Research Department I had an encounter with matrix computations which puzzled me a good deal. After a succession of failures I had been presented with a system of twelve linear equations to solve. I was delighted at last at being given a problem which I "knew all about" and had departed with my task, confident that I would return with the solution in a very short time. However, when I returned to my room my confidence rapidly evaporated. The set of 144 coefficients suddenly looked very much larger than they had seemed when I was given them. I consulted the few books that were then available, one of which, incidentally, recommended the direct application of Cramer's

rule using determinants! It did not take long to appreciate that this was not a good idea and I finally decided to use Gaussian elimination with what would now be called "partial pivoting."

Anxiety about rounding errors in elimination methods had not yet reared its head and I used ten-decimal computation more as a safety precaution rather than because I was expecting any severe instability problems. The system was mildly ill-conditioned, though we were not so free with such terms of abuse in those days, and starting from coefficients of order unity, I slowly lost figures until the final reduced equation was of the form, say,

$$.0000376235x_{12} = .0000216312$$

At this stage I can remember thinking to myself that the computed x_{12} derived from this relation could scarcely have more than six correct figures, even supposing that there had been no buildup in rounding errors, and I contemplated computing the answers to six figures only. However, as those of you who have worked with a desk computer will know, one tends to make fewer blunders if one adheres to a steady pattern of work, and accordingly I computed all variables to ten figures, though fully aware of the absurdity of doing so. It so happened that all solutions were of order unity, which from the nature of the physical problem was to be expected.

Then, being by that time a well-trained computer, I substituted my solution in the original equations to see how they checked. Since x_1 had been derived from the first of the original equations, I started by substituting in the 12th equation. You will appreciate that on a desk machine the inner-product is accumulated exactly giving 20 figures in all. (It is interesting that nowadays we nearly always accept a poorer performance from the arithmetic units of computers!) To my astonishment the left-hand side agreed with the given right-hand side to ten figures, i.e. to the full extent of the righthand side. That, I said to myself, was a coincidence. Eleven more "coincidences" followed, though perhaps not quite in rapid succession! I was completely baffled by this. I felt sure that none of the variables could have more than six correct figures and yet the agreement was as good as it would have been if I had been given the exact answer and had then rounded it to ten figures. However, the war had still to be won, and it was no time to become introspective about rounding errors; in any case I had already taken several times longer than my first confident estimate. My taskmaster was not as appreciative as he might have been but he had to admit he was impressed when I claimed that I had "the exact solution" corresponding to a right-hand side which differed only in the tenth figure from the given one.

As you can imagine this experience was very much in my mind when I arrived at N.P.L. and encountered the preoccupation with the instability of elimination methods. Of course I still believed that my computed answers had at best six correct figures, but it was puzzling that in my only encounter with linear systems it was the surprising *accuracy* of the solutions (at least in the sense of small residuals) which required an explanation. In the current climate at N.P.L. I decided not to risk looking foolish by stressing this experience.

However, it happened that some time after my arrival, a system of 18 equations arrived in Mathematics Division and after talking around it for some time we finally decided to abandon theorizing and to solve it. A system of 18 is surprisingly formidable, even when one has had previous experience with 12, and we accordingly de-

cided on a joint effort. The operation was manned by Fox, Goodwin, Turing, and me, and we decided on Gaussian elimination with complete pivoting. Turing was not particularly enthusiastic, partly because he was not an experienced performer on a desk machine and partly because he was convinced that it would be a failure. History repeated itself remarkably closely. Again the system was mildly ill-conditioned, the last equation had a coefficient of order 10^{-4} (the original coefficients being of order unity) and the residuals were again of order 10^{-10} , that is of the size corresponding to the exact solution rounded to ten decimals. It is interesting that in connection with this example we subsequently performed one or two steps of what would now be called "iterative refinement," and this convinced us that the first solution had had almost six correct figures.

I suppose this must be regarded as a defeat for Turing since he, at that time, was a keener adherent than any of the rest of us to the pessimistic school. However, I'm sure that this experience made quite an impression on him and set him thinking afresh on the problem of rounding errors in elimination processes. About a year later he produced his famous paper "Rounding-off errors in matrix processes" [1] which together with the paper of J. von Neumann and H. Goldstine [4] did a great deal to dispel the gloom. The second round undoubtedly went to Turing!

This anecdote illustrates rather well, I think, the confused state of mind which existed at that time, and was shared even by the most distinguished people working in the field. By contrast I think we can fairly claim today to have a reasonably complete understanding of matrix stability problems, not only in the solution of linear systems, but also in the far more difficult eigenvalue problem.

Failures in the Matrix Field

Although we can claim to have been successful in the matrix area as far as the development of algorithms and an understanding of their performance is concerned, there are other respects in which we have not been particularly successful even in this field. Most important of these is a partial failure in communication. The use of algorithms and a general understanding of the stability problem has lagged much further behind developments than it should have. The basic problems of matrix computation have the advantage of simple formulations, and I feel that the preparation of well-tested and well-documented algorithms should have advanced side by side with their development and analysis. There are two reasons why this has not happened. (i) It is a much more arduous task than was appreciated to prepare the documentation thoroughly. (ii) Insufficient priority has been attached to doing it. There are signs in the last year or two that these shortcomings are at last being overcome with the work on the *Handbook for Automatic Computation* [5], that on matrix algorithms centered at Argonne National Laboratory, and the more general project at Bell Telephone Laboratories [6]. I think it is of vital importance that all the work that has been expended on the development of satisfactory algorithms should be made fully available to the people who need to use it. I would go further than this and claim that it is a social duty to see that this is achieved.

A second disquieting feature about work in the matrix field is that it has tended to be isolated from that in very closely related areas. I would like to mention in particular linear programming and statistical computations. Workers in linear algebra and linear programming seemed until recently to comprise almost com-

pletely disjoint sets and this is surely undesirable. The standard computations required in practical statistics provide the most direct opportunities for applying the basic matrix algorithms and yet there is surprisingly little collaboration. Only recently I saw an article by a well-known practical statistician on the singular value decomposition which did not, at least in its first draft, contain any reference to the work of Kahan and Golub who have developed such an admirable algorithm for this purpose. Clearly there is a failure on both sides, but I think it is primarily the duty of people working in the matrix field to make certain that their work is used in related areas, and this calls for an aggressive policy. Again there are signs that this isolation is breaking down. At Stanford, Professor Dantzig, a pioneer in linear programming, now has a joint appointment with the Computer Science Department and schemes are afoot in the UK to have joint meetings of matrix experts and members of the Statistical Society. Historical accidents often play a great part in breaking down barriers and it is interesting that collaboration between workers on the numerical solution of partial differential equations and on matrix algebra has always been extremely close.

A third disappointing feature is the failure of numerical analysts to influence computer hardware and software in the way that they should. In the early days of the computer revolution computer designers and numerical analysts worked closely together and indeed were often the same people. Now there is a regrettable tendency for numerical analysts to opt out of any responsibility for the design of the arithmetic facilities and a failure to influence the more basic features of software. It is often said that the use of computers for scientific work represents a small part of the market and numerical analysts have resigned themselves to accepting facilities "designed" for other purposes and making the best of them. I am not convinced that this is inevitable, and if there were sufficient unity in expressing their demands there is no reason why they could not be met. After all, one of the main virtues of an electronic computer from the point of view of the numerical analyst is its ability to "do arithmetic fast." Need the arithmetic be so bad! Even here there are hopeful developments. The work of W. Kahan deserves particular mention and last September a well-known manufacturer sponsored a meeting on this topic at which he, among others, had an opportunity to express his views.

Final Comments

I am convinced that mathematical computation has a great part to play in the future and that its contribution will fully live up to the expectations of the great pioneers of the computer revolution. The greatest danger to numerical analysts at the moment springs from a lack of faith in themselves for which there is no real justification. I think the nature of research in numerical analysis is bound to change substantially in the next decade. In the first two decades we have concentrated on the basic problems, such as arise, for example, in linear and nonlinear algebra and approximation theory. In retrospect these will appear as a preliminary sharpening of the tools which we require for the real task. For success in this it will be essential to recruit more effectively than we have so far from the ranks of applied mathematicians and mathematical physicists. On a recent visit to the Soviet Union I was struck by the fact that most of the research in numerical analysis is being done

by people who were essentially mathematical physicists, who have decided to tackle their problems by numerical methods, and they are strongly represented in the Academy of Sciences. Although I think that we in the West have nothing to fear from a comparison of achievements, I do feel that morale is markedly higher in the Soviet Union.

In the UK there are signs that the tide is already turning. There is to be a Numerical Analysis Year at the University of Dundee, during the course of which many of the more distinguished of the world's numerical analysts will be visiting the UK. Quite recently a Discussion Meeting on a numerical analysis topic was held at the Royal Society. Such things would scarcely have been contemplated a year or two ago. I look forward to the time when numerical mathematics will dominate the applied field and will again occupy a central position at meetings of the ACM.

REFERENCES

1. TURING, A. M. Rounding-off errors in matrix processes. *Quart. J. Mech.*, 1 (1948), 287-308.
2. DAVIS, P. J. Numerical analysis. In *The Mathematical Sciences: A Collection of Essays*. MIT Press, Cambridge, Mass., 1969.
3. FADDEEVA, V. N. *Computational Methods of Linear Algebra*, Translated by C. D. Benster. Dover, New York, 1959.
4. VON NEUMANN, J. AND GOLDSTINE, H. H. Numerical inverting of matrices of high order. *Bull. Amer. Math. Soc.* 53 (1947), 1021-1099.
5. WILKINSON, J. H. *Handbook for Automatic Computation, Vol. 2. Linear Algebra*. Springer-Verlag, Berlin (to be published).
6. GENTLEMAN, W. M., AND TRAUB, J. F. The Bell Laboratories numerical mathematics program library project. Proc. ACM 23rd Nat. Conf., 1968, Brandon/Systems Press, Princeton, N. J., pp. 485-490.

Logic and Programming Languages

Dana S. Scott
University of Oxford

Logic has been long interested in whether answers to certain questions are computable in principle, since the outcome puts bounds on the possibilities of formalization. More recently, precise comparisons in the efficiency of decision methods have become available through the developments in complexity theory. These, however, are applications to logic, and a big question is whether methods of logic have significance in the other direction for the more applied parts of computability theory.

Programming languages offer an obvious opportunity as their syntactic formalization is well advanced; however, the semantical theory can hardly be said to be complete. Though we have many examples, we have still to give wide-ranging mathematical answers to these queries: What is a machine? What is a computable process? How (or how well) does a machine simulate a process? Programs naturally enter in giving descriptions of processes. The definition of the precise meaning of a program then requires us to explain what are the objects of computation (in a way, the statics of the problem) and how they are to be transformed (the dynamics).

So far the theories of automata and of nets, though most interesting for dynamics, have formalized only a

portion of the field, and there has been perhaps too much concentration on the finite-state and algebraic aspects. It would seem that the understanding of higher-level program features involves us with infinite objects and forces us to pass through several levels of explanation to go from the conceptual ideas to the final simulation on a real machine. These levels can be made mathematically exact if we can find the right abstractions to represent the necessary structures.

The experience of many independent workers with the method of data types as lattices (or partial orderings) under an information content ordering, and with their continuous mappings, has demonstrated the flexibility of this approach in providing definitions and proofs, which are clean and without undue dependence on implementations. Nevertheless much remains to be done in showing how abstract conceptualizations can (or cannot) be actualized before we can say we have a unified theory.

Key Words and Phrases: logic, programming languages, automata, denotational semantics, λ -calculus models, computability, partial functions, approximation, function spaces

CR Categories: 1.2, 4.20, 5.21, 5.24, 5.27

As the eleven-and-one-half-th Turing lecturer, it gives me the greatest pleasure to share this prize and this podium with Michael Rabin. Alas, we have not had much chance to collaborate since the time of writing our 1959 paper, and that is for me a great loss. I work best in collaboration, but it is not easy to arrange the right conditions—especially in interdisciplinary subjects and where people are separated by international boundaries. But I have followed his career with deep interest and admiration. As you have heard today, Rabin has been able to *apply* ideas from logic having to do with decidability, computability and complexity to

questions of real mathematical and computational interest. He, and many others, are actively creating new methods of analysis for a wide class of algorithmic problems which has great promise for future development. These aspects of the theory of computation are, however, quite outside my competence, since over the years my interests have diverged from those of Rabin. From the late 1960's my own work has concentrated on seeing whether the ideas of logic can be used to give a better *conceptual* understanding of programming languages. I shall therefore not speak today in detail about my past joint work with Rabin but about my own development and some plans and hopes for the future.

The difficulty of obtaining a precise overall view of a language arose during the period when committees were constructing mammoth "universal" computer languages. We stand now, it seems, on the doorstep of yet another technological revolution during which our ideas of machines and software are going to be completely changed. (I have just noted that the ACM is

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's Address: Mathematical Institute, University of Oxford, 24-29 St. Giles, Oxford OX1 3LB, U.K.

campaigning again to eliminate the word 'machine' altogether.) The big, big languages may prove to be not very adaptable, but I think the problem of *semantics* will surely remain. I would like to think that the work—again done in collaboration with other people, most notably with the late Christopher Strachey—has made a basic contribution to the foundations of the semantic enterprise. Well, we shall see. I hope too that the research on semantics will not too much longer remain disjoint from investigations like Rabin's.

An Apology and a Nonapology

As a rule, I think, public speakers should not apologize: it only makes the audience uncomfortable. At such a meeting as this, however, one apology is necessary (along with a disclaimer).

Those of you who know my background may well be reminded of Sir Nicholas Gimcrack, hero of the play *The Virtuoso*. It was written in 1676 by Thomas Shadwell to poke a little fun at the remarkable experiments then being done before the Royal Society of London. At one point in the play, Sir Nicholas is discovered lying on a table trying to learn to swim by imitating the motions of a frog in a bowl of water. When asked whether *he* had ever practiced swimming *in water*, he replies that he hates water and would never go near it! "I content myself," he said, "with the speculative part of swimming; I care not for the practical. I seldom bring anything to use. . . . Knowledge is the ultimate end."

Now though our ultimate aims are the same, I hasten to disassociate myself from the attitude of disdain for the practical. It is, however, the case that I have no practical experience in present-day programming; by necessity I have had to confine myself to speculative programming, gaining what knowledge I could at second hand by watching various frogs and other creatures. Luckily for me, some of the frogs could speak. With some of them I have had to learn an alien language, and perhaps I have not understood what they were about. But I have *tried* to read and to keep up with developments. I apologize for not being a professional in the programming field, and I certainly, therefore, will not try to sermonize: many of the past Turing lecturers were well equipped for that, and they have given us very good advice. What I try to do is to make some results from logic which seem to me to be relevant to computing *comprehensible* to those who could make use of them. I have also tried to add some results of my own, and I have to leave it to you to judge how successful my activities have been.

Most fortunately today I do *not* have to apologize for the lack of published material; if I had written this talk the day I received the invitation, I might have. But in the August number of *Communications* we have the excellent tutorial paper by Robert Tennent [14] on denotational semantics, and I very warmly recommend

it as a starting place. Tennent not only provides serious examples going well beyond what Strachey and I ever published, but he also has a well-organized bibliography.

Only last month the very hefty book by Milne and Strachey [9] was published. Strachey's shockingly sudden and untimely death unfortunately prevented him from ever starting on the revision of the manuscript. We have lost much in style and insight (to say nothing of inspiration) by Strachey's passing, but Robert Milne has carried out their plan admirably. What is important about the book is that it pushes the discussion of a complex language through from the *beginning* to the *end*. Some may find the presentation too rigorous, but the point is that the semantics of the book is not mere speculation but the real thing. It is the product of serious and informed thought; thus, one has the detailed evidence to decide whether the approach is going to be fruitful. Milne has organized the exposition so one can grasp the language on many levels down to the final compiler. He has not tried to sidestep any difficulties. Though not lighthearted and biting, as Strachey often was in conversation, the book is a very fitting memorial to the last phase of Strachey's work, and it contains any number of original contributions by Milne himself. (I can say these things because I had no hand in writing the book myself.)

Recently published also is the volume by Donahue [4]. This is a not too long and very readable work that discusses issues not covered, or not covered from the same point of view, by the previously mentioned references. Again, it was written quite independently of Strachey and me, and I was very glad to see its appearance.

Soon to come out is the textbook by Joe Stoy [13]. This will complement these other works and should be very useful for teaching, because Stoy has excellent experience in lecturing, both at Oxford University and at M.I.T.

On the foundational side, my own revised paper (Scott [12]) will be out any moment in the *SIAM Journal on Computing*. As it was written from the point of view of enumeration operators in more "classical" recursion theory, its relevance to practical computing may not be at all clear at first glance. Thus I am relieved that these other references explain the uses of the theory in the way I intended.

Fortunately all the above authors cite the literature extensively, and so I can neglect going into further historical detail today. May I only say that many other people have taken up various of the ideas of Strachey and myself, and you can find out about their work not only from these bibliographies but also, for example, from two recent conference proceedings, Manes [7] and Böhm [1]. If I tried to list names here, I would only leave some out—those that have had contact with me know how much I appreciate their interest and contributions.

Some Personal Notes

I was born in California and began my work in mathematical logic as an undergraduate at Berkeley in the early 1950's. The primary influence was, of course, Alfred Tarski together with his many colleagues and students at the University of California. Among many other things, I learned recursive function theory from Raphael and Julia Robinson, whom I want to thank for numerous insights. Also at the time through self-study I found out about the λ -calculus of Curry and Church (which, literally, gave me nightmares at first). Especially important for my later ideas was the study of Tarski's semantics and his definition of truth for formalized languages. These concepts are still being hotly debated today in the philosophy of natural language, as you know. I have tried to carry over the spirit of Tarski's approach to algorithmic languages, which at least have the advantage of being reasonably well formalized syntactically. Whether I have found the *right* denotations of terms as guided by the schemes of Strachey (and worked out by many hands) is what needs discussion. I am the first to say that not *all* problems are solved just by giving denotations to *some* languages. Languages like (the very pure) λ -calculus are well served, but many programming concepts are still not covered.

My graduate work was completed in Princeton in 1958 under the direction of Alonzo Church, who also supervised Michael Rabin's thesis. Rabin and I met at that time, but it was during an IBM summer job in 1957 that we did our joint work on automata theory. It was hardly carried out in a vacuum, since many people were working in the area; but we did manage to throw some basic ideas into sharp relief. At the time I was certainly thinking of a project of giving a mathematical definition of a machine. I feel now that the finite-state approach is only partially successful and without much in the way of practical implication. True, many physical machines can be modelled as finite-state devices; but the *finiteness* is hardly the most important feature, and the automata point of view is often rather superficial.

Two later developments made automata seem to me more interesting, at least mathematically: the Chomsky hierarchy and the connections with semigroups. From the algebraic point of view (to my taste at least) Eilenberg, the Euclid of automata theory, in his books [5] has said pretty much the last word. I note too that he has avoided abstract category theory. Categories may lead to good things (cf. Manes [7]), but too early a use can only make things too difficult to understand. That is my personal opinion.

In some ways the Chomsky hierarchy is in the end disappointing. Context-free languages are very important and everyone has to learn about them, but it is not at all clear to me what comes next—if anything. There are so many other families of languages, but not much order has come out of the chaos. I do not think the last

word has been said here. It was not knowing where to turn, and being displeased with what I thought was excessive complexity, that made me give up working in automata theory. I tried once in a certain way to connect automata and programming languages by suggesting a more systematic way of separating the machine from the program. Eilenberg heartily disliked the idea, but I was glad to see the recent book by Clark and Cowell [2] where, at the suggestion of Peter Landin, the idea is carried out very nicely. It is not algebra, I admit, but it seems to me to be (elementary, somewhat theoretical) programming. I would like to see the next step, which would fall somewhere in between Manna [8] and Milne-Strachey [9].

It was at Princeton that I had my first introduction to real machines—the now almost prehistoric von Neumann machine. I have to thank Forman Acton for that. Old fashioned as it seems now, it was still *real*; and Hale Trotter and I had great fun with it. How very sad I was indeed to see the totally dead corpse in the Smithsonian Museum with no indication at all what it was like when it was alive.

From Princeton I went to the University of Chicago to teach in the Mathematics Department for two years. Though I met Bob Ashenhurst and Nick Metropolis at that time, my stay was too short to learn from them; and as usual there is always too great a distance between departments. (Of course, since I am only writing about connections with computing, I am not trying to explain my other activities in mathematics and logic.)

From Chicago I went to Berkeley for three years. There I met many computer people through Harry Huskey and René de Vogelaere, the latter of whom introduced me to the details of Algol 60. There was, however, no Computer Science Department as such in Berkeley at that time. For personal reasons I decided soon to move to Stanford. Thus, though I taught a course in Theory of Computation at Berkeley for one semester, my work did not amount to anything. One thing I shall always regret about Berkeley and Computing is that I never learned the details of the work of Dick and Emma Lehmer, because I very much admire the way they get *results* in number theory by machine. Now that we have the Four-Color Problem solved by machine, we are going to see great activity in large-scale, special-purpose theorem proving. I am very sorry not to have any hand in it.

Stanford had from the early 1960's one of the best Computer Science departments in the country, as everyone agrees. You will wonder why I ever left. The answer may be that my appointment was a mixed one between the departments of Philosophy and Mathematics. I suppose my *personal* difficulty is knowing where I should be and what I want to do. But personal failings aside, I had excellent contacts in Forsythe's remarkable department and very good relations with the graduates, and we had many lively courses and seminars. John McCarthy and Pat Suppes, and people

from their groups, had much influence on me and my views of computing. In Logic, with my colleagues Sol Feferman and Georg Kreisel, we had a very active group. Among the many Ph.D. students in Logic, the work of Richard Platek had a few years later, when I saw how to use some of his ideas, much influence on me.

At this point I had a year's leave in Amsterdam which proved unexpectedly to be a turning point in my intellectual development. I shall not go into detail, since the story is complicated; but the academic year 1968/69 was one of deep crisis for me, and it is still very painful for me to think back on it. As luck would have it, however, Pat Suppes had proposed my name for the IFIP Working Group 2.2 (now called Formal Description of Programming Concepts). At that time Tom Steel was Chairman, and it was at the Vienna meeting that I first met Christopher Strachey. If the violence of the arguments in this group are any indication, I am really glad I was not involved with anything important like the Algol Committee. But I suppose fighting is therapeutic: it brings out the best and the worst in people. And in any case it is good to learn to defend oneself. Among the various combatants I liked the style and ideas of Strachey best, though I think he often overstated his case; but what he said convinced me I should learn more.

It was only at the end of my year in Amsterdam that I began to talk with Jaco de Bakker, and it was only through correspondence over that summer that our ideas took definite shape. The Vienna IBM Group that I met through WG 2.2 influenced me at this stage also. In the meantime I had decided to leave Stanford for the Princeton Philosophy Department; but since I was in Europe with my family, I requested an extra term's leave so I could visit Strachey in Oxford in the fall of 1969. That term was one of feverish activity for me; indeed, for several days, I felt as though I had some kind of real brain fever. The collaboration with Strachey in those few weeks was one of the best experiences in my professional life. We were able to repeat it once more the next summer in Princeton, though at a different level of excitement. Sadly, by the time I came to Oxford permanently in 1972, we were both so involved in teaching and administrative duties that real collaboration was nearly impossible. Strachey also became very discouraged over the continuing lack of research funds and help in teaching, and he essentially withdrew himself to write his book with Milne. (It was a great effort and I do not think it did his health any good; how I wish he could have seen it published.)

Returning to 1969, what I started to do was to show Strachey that he was *all wrong* and that he ought to do things in quite another way. He had originally had his attention drawn to the λ -calculus by Roger Penrose and had developed a handy style of using this notation for functional abstraction in explaining programming concepts. It was a *formal* device, however, and I tried to

argue that it had *no* mathematical basis. I have told this story before, so to make it short, let me only say that in the first place I had actually convinced him by "superior logic" to give up the type-free λ -calculus. But then, as one consequence of my suggestions followed the other, I began to see that computable functions could be defined on a great variety of spaces. The real step was to see that function-spaces were good spaces, and I remember quite clearly that the logician Andrzej Mostowski, who was also visiting Oxford at the time, simply did not believe that the kind of function spaces I defined had a constructive description. But when I saw they actually did, I began to suspect that the possibilities of using function spaces might just be more surprising than we had supposed. Once the doubt about the enforced rigidity of logical types that I had tried to push onto Strachey was there, it was not long before I had found one of the spaces isomorphic with its own function space, which provides a model of the "type-free" λ -calculus. The rest of the story is in the literature.

(An interesting sidelight on the λ -calculus is the rôle of Alan Turing. He studied at Princeton with Church and connected computability with the (formal) λ -calculus around 1936/37. Illuminating details of how his work (and the further influence of λ -calculus) was viewed by Steve Kleene can be found in Crossley [3]. (Of course Turing's later ideas about computers very much influenced Strachey, but this is not the time for a complete historical analysis.) Though I never met Turing (he died in 1954), the second-hand connections through Church and Strachey and my present Oxford colleagues, Les Fox and Robin Gandy, are rather close, though by the time I was a graduate at Princeton, Church was no longer working on the λ -calculus, and we never discussed his experiences with Turing.)

It is very strange that my λ -calculus models were not discovered earlier by someone else; but I am most encouraged that new kinds of models with new properties are now being discovered, such as the "powerdomains" of Gordon Plotkin [10]. I am personally convinced that the field is well established, both on the theoretical and on the applied side. John Reynolds and Robert Milne have independently introduced a new inductive method of proving equivalences, and the interesting work of Robin Milner on LCF and its proof techniques continues at Edinburgh. This direction of proving things about models was started off by David Park's theorem on relating the fixed-point operator and the so-called paradoxical combinator of λ -calculus, and it opened up a study of the infinitary, yet computable operators which continues now along many lines. Another direction of work goes on in Novosibirsk under Yu.L. Ershov, and quite surprising connections with topological algebra have been pointed out to me by Karl H. Hofmann and his group. There is no space here even to begin to list the many contributors.

In looking forward to the next few years, I am particularly happy to report at this meeting that Tony

Hoare has recently accepted the Chair of Computation at Oxford, now made permanent since Strachey's passing. This opens up all sorts of new possibilities for collaboration, both with Hoare and with the many students he will attract after he takes up the post next year. And, as you know, the practical aspects of use and design of computer languages and of programming methodology will certainly be stressed at Oxford (as Strachey did too, I hasten to add), and this is all to the good; but there is also excellent hope for theoretical investigations.

Some Semantic Structures

Turning now to technical details, I should like to give a brief indication of how my construction goes, and how it is open to considerable variation. It will not be possible to argue here that these are the "right" abstractions, and that is why it is a relief to have those references mentioned earlier so easily available.

Perhaps the quickest indication of what I am getting at is provided by two domains: \mathcal{B} , the domain of Boolean values, and $\mathcal{S} = \mathcal{B}^\omega$, the domain of infinite sequences of Boolean values. The first main point is that we are going to accept the idea of partial functions represented mathematically by giving the functions from time to time partial values. As far as \mathcal{B} goes the idea is very trivial: we write

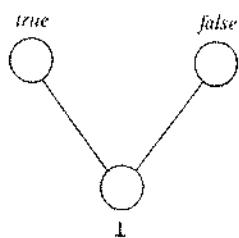
$$\mathcal{B} = \{\text{true}, \text{false}, \perp\}$$

where \perp is an extra element called "the undefined." In order to keep \perp in its place we impose a partial ordering \sqsubseteq on the domain \mathcal{B} , where

$$x \sqsubseteq y \text{ iff either } x = \perp \text{ or } x = y,$$

for all $x, y \in \mathcal{B}$. It will not mean all that much here in \mathcal{B} , but we can read " \sqsubseteq " as saying that the information content of x is contained in the information content of y . The element \perp has, therefore, empty information content. The scheme is illustrated in Figure 1.

Fig. 1. The Boolean values.



(An aside: in many publications I have advocated using lattices, which as partial orderings have a "top" element T as well as a "bottom" element \perp , so that we can assert $\perp \sqsubseteq x \sqsubseteq T$ for all elements of the domain. This suggestion has not been well received for many reasons I cannot go into here. Some discussion of its reasona-

bility is to be found in Scott [12], but of course the structure studied there is special. Probably it is best neither to exclude or include a \perp ; and, for simplicity, I shall not mention it further today.)

Looking now at \mathcal{S} , the domain of sequences, we shall employ a shorthand notation where subscripts indicate the coordinates; thus,

$$x = \langle x_n \rangle_{n=0}^{\infty}$$

for all $x \in \mathcal{S}$. Each term is such that $x_n \in \mathcal{B}$, because $\mathcal{S} = \mathcal{B}^\omega$. Technically, a "direct product" of structures is intended, so we define \sqsubseteq on \mathcal{S} by

$$x \sqsubseteq y \text{ iff } x_n \sqsubseteq y_n, \text{ for all } n.$$

Intuitively, a sequence y is "better" in information than a sequence x iff some of the coordinates of x which were "undefined" have passed over into "being defined" when we go from x to y . For example, each of the following sequences stands in the relation \sqsubseteq to the following ones:

$$\begin{aligned} &\langle \perp, \perp, \perp, \perp, \dots \rangle, \\ &\langle \text{true}, \perp, \perp, \perp, \dots \rangle, \\ &\langle \text{true}, \text{false}, \perp, \perp, \dots \rangle, \\ &\langle \text{true}, \text{false}, \text{true}, \perp, \dots \rangle. \end{aligned}$$

Clearly this list could be expanded infinitely, and there is also no need to treat the coordinates in the strict order $n = 0, 1, 2, \dots$. Thus the \sqsubseteq relation on \mathcal{S} is far more complex than the original \sqsubseteq on \mathcal{B} .

An obvious difference between \mathcal{B} and \mathcal{S} is that \mathcal{B} is finite while \mathcal{S} has infinitely many elements. In \mathcal{S} , also, certain elements have infinite information content, whereas this is not so in \mathcal{B} . However, we can employ the partial ordering in \mathcal{S} to explain abstractly what we mean by "finite approximation" and "limits." The sequences listed above are finite in \mathcal{S} because they have only finitely many coordinates distinct from \perp . Given any $x \in \mathcal{S}$ we can cut it down to a finite element by defining

$$(x \upharpoonright m)_n = \begin{cases} x_n, & \text{if } n < m; \\ \perp, & \text{if not.} \end{cases}$$

It is easy to see from our definitions that

$$x \upharpoonright m \sqsubseteq x \upharpoonright (m + 1) \sqsubseteq x,$$

so that the $x \upharpoonright m$ are "building up" to a limit; and, in fact, that limit is the original x . We write this as

$$x = \bigcup_{m=0}^{\infty} (x \upharpoonright m),$$

where \bigcup is the sup or least-upper-bound operation in the partially ordered set \mathcal{S} . The point is that \mathcal{S} has many supers; and, whenever we have elements $y^{(m)} \sqsubseteq y^{(m+1)}$ in \mathcal{S} (regardless of whether they are finite or not), we can define the "limit" z , where

$$z = \bigcup_{m=0}^{\infty} y^{(m)}.$$

(Hint: ask yourself what the coordinates of z will have to be.) We cannot rehash the details here, but \mathcal{S} really is a topological space, and z really is a limit. Thus, though \mathcal{S} is infinitary, there is a good chance that we can let manipulations fall back on finitary operations and be able to discuss *computable* operations on \mathcal{S} and on more complex domains.

Aside from the sequence and partial-order structure on \mathcal{S} , we can define many kinds of algebraic structure. That is why \mathcal{S} is a good example. For instance, up to isomorphism the space satisfies

$$\mathcal{S} = \mathcal{S} \times \mathcal{S},$$

where on the right-hand side the usual binary direct product is intended. Abstractly, the domain $\mathcal{S} \times \mathcal{S}$ consists of all ordered pairs $\langle x, y \rangle$ with $x, y \in \mathcal{S}$, where we define \sqsubseteq on $\mathcal{S} \times \mathcal{S}$ by

$$\langle x, y \rangle \sqsubseteq \langle x', y' \rangle \text{ iff } x \sqsubseteq x' \text{ and } y \sqsubseteq y'.$$

But for all practical purposes there is no harm in identifying $\langle x, y \rangle$ with a sequence already in \mathcal{S} ; indeed coordinatewise we can define

$$\langle x, y \rangle_n = \begin{cases} x_k, & \text{if } n = 2k; \\ y_k, & \text{if } n = 2k + 1. \end{cases}$$

The above criterion for \sqsubseteq between pairs will be verified, and we can say that \mathcal{S} has a (bi-unique) pairing function.

The pairing function $\langle \cdot, \cdot \rangle$ on \mathcal{S} has many interesting properties. In effect we have already noted that it is *monotonic* (intuitively: as you increase the information contents of x and y , you increase the information content of $\langle x, y \rangle$.) More importantly, $\langle \cdot, \cdot \rangle$ is *continuous* in the following precise sense:

$$\langle x, y \rangle = \bigcup_{m=0}^{\infty} \langle x \upharpoonright m, y \upharpoonright m \rangle$$

which means that $\langle \cdot, \cdot \rangle$ behaves well under taking finite approximations. And this is only one example; the whole *theory* of monotone and continuous functions is very important to this approach.

Even with the small amount of structure we have put on \mathcal{S} , a *language* suggests itself. For the sake of illustration, we concentrate on the two isomorphisms satisfied by \mathcal{S} ; namely, $\mathcal{S} = \mathcal{B} \times \mathcal{S}$ and $\mathcal{S} = \mathcal{S} \times \mathcal{S}$. The first identified \mathcal{S} as having to do with (infinite) sequences of Boolean values; while the second reminds us of the above discussion of the pairing function. In Figure 2 we set down a quick BNF definition of a language with two kinds of expressions: *Boolean* (the β 's) and *sequential* (the σ 's).

Fig. 2. A brief language.

```

 $\beta ::= \text{true} \mid \text{false} \mid \text{head } \sigma$ 
 $\sigma ::= \beta^* \mid \beta\sigma \mid \text{tail } \sigma \mid$ 
       $\text{if } \beta \text{ then } \sigma' \text{ else } \sigma'' \mid$ 
       $\text{even } \sigma \mid \text{odd } \sigma \mid \text{merge } \sigma'\sigma''$ 

```

This language is very brief indeed: no variables, no declarations, no assignments, only a miniature selec-

tion of constant terms. Note that the notation chosen was meant to make the meanings of these expressions obvious. Thus, if σ denotes a sequence x , then **head** σ has got to denote the first term x_0 of the sequence x . As $x_0 \in \mathcal{B}$ and $x \in \mathcal{S}$, we are keeping our types straight.

More precisely, for each expression we can define its (constant) *value* $\llbracket \cdot \rrbracket$; so that $\llbracket \beta \rrbracket \in \mathcal{B}$ for Boolean expressions β , and $\llbracket \sigma \rrbracket \in \mathcal{S}$ for sequential expressions. Since there are ten clauses in the BNF language definition, we would have to set down ten equations to completely specify the semantics of this example; we shall content ourselves with selected equations here. To carry on with the remark in the last paragraph:

$$\llbracket \text{head } \sigma \rrbracket = \llbracket \sigma \rrbracket_0.$$

On the other side, the expression β^* creates an infinite sequence of Boolean values:

$$\llbracket \beta^* \rrbracket = \langle \llbracket \beta \rrbracket, \llbracket \beta \rrbracket, \llbracket \beta \rrbracket, \llbracket \beta \rrbracket, \dots \rangle.$$

(This notation, though rough, is clear.) In the same vein:

$$\llbracket \beta\sigma \rrbracket = \langle \llbracket \beta \rrbracket, \llbracket \sigma \rrbracket_0, \llbracket \sigma \rrbracket_1, \llbracket \sigma \rrbracket_2, \dots \rangle;$$

while we have

$$\llbracket \text{tail } \sigma \rrbracket = \langle \llbracket \sigma \rrbracket_1, \llbracket \sigma \rrbracket_2, \llbracket \sigma \rrbracket_3, \llbracket \sigma \rrbracket_4, \dots \rangle.$$

Further along:

$$\llbracket \text{even } \sigma \rrbracket = \langle \llbracket \sigma \rrbracket_0, \llbracket \sigma \rrbracket_2, \llbracket \sigma \rrbracket_4, \llbracket \sigma \rrbracket_6, \dots \rangle;$$

and

$$\llbracket \text{merge } \sigma'\sigma'' \rrbracket = \langle \llbracket \sigma' \rrbracket, \llbracket \sigma'' \rrbracket \rangle.$$

These should be enough to give the idea. It should also be clear that what we have is really only a selection, because \mathcal{S} satisfies many more isomorphisms (e.g., $\mathcal{S} = \mathcal{S} \times \mathcal{S} \times \mathcal{S}$), and there are many, many more ways of tearing apart and recombining sequences of Boolean values—all in quite computable ways.

The Function Space

It should not be concluded that the previous section contains the whole of my idea; this would leave us on the elementary level of program schemes (e.g. van Emden-Kowalski [6] or Manna [8] (last chapter)). What some people call “Fixpoint Semantics” (I myself do not like the abbreviated word “fixpoint”) is only a *first chapter*. The *second chapter* already includes procedures that take procedures as arguments—higher-type procedures—and we are well beyond program schemes. True, fixed-point techniques can be applied to these higher-type procedures, but that is not the only thing to say in their favor. The semantic structure needed to make this definite is the *function space*. I

have tried to stress this from the start in 1969, but many people have not understood me well enough.

Suppose \mathcal{S}' and \mathcal{S}'' are two domains of the kind we have been discussing (say, \mathcal{B} or $\mathcal{B} \times \mathcal{B}$ or \mathcal{S} or something worse). By $[\mathcal{S}' \rightarrow \mathcal{S}']$ let us understand the domain of all monotone and continuous functions f mapping \mathcal{S}' into \mathcal{S}'' . This is what I mean by a *function space*. It is not all that difficult mathematically, but it is not all that obvious either that $[\mathcal{S}' \rightarrow \mathcal{S}']$ is again a domain "of the same kind," though admittedly of a more complicated structure. I cannot prove it here, but at least I can define the \sqsubseteq relation on the function space:

$$f \sqsubseteq g \text{ iff } f(x) \sqsubseteq g(x) \text{ for all } x \in \mathcal{S}'.$$

Treating functions as abstract *objects* is nothing new; what has to be checked is that they are also quite reasonable *objects of computation*. The relation \sqsubseteq on $[\mathcal{S}' \rightarrow \mathcal{S}']$ is the first step in checking this, and it leads to a well-behaved notion of a finite approximation to a function. (Sorry! there is no time to be more precise here.) And when that is seen, the way is open to *iteration* of function spaces; as in $[[\mathcal{S}' \rightarrow \mathcal{S}'] \rightarrow \mathcal{S}''']$. This is not as crazy as it might seem at first, since our theory identifies $f(x)$ as a computable binary function of *variable* f and *variable* x . Thus, as an operation, it can be seen as an *element* of a function space:

$$[[[\mathcal{S}' \rightarrow \mathcal{S}'] \times \mathcal{S}'] \rightarrow \mathcal{S}''].$$

This is only the start of a theory of these operators (or *combinators*, as Curry and Church call them).

Swallowing all this, let us attempt an infinite iteration of function spaces beginning with \mathcal{S} . We define $\mathcal{F}_0 = \mathcal{S}$ and $\mathcal{F}_{n+1} = [\mathcal{F}_n \rightarrow \mathcal{S}]$. Thus $\mathcal{F}_1 = [\mathcal{S} \rightarrow \mathcal{S}]$ and $\mathcal{F}_4 = [[[[\mathcal{S} \rightarrow \mathcal{S}] \rightarrow \mathcal{S}] \rightarrow \mathcal{S}] \rightarrow \mathcal{S}]$.

You just have to believe me that this is all highly constructive (because we employ only the continuous functions).

It is fairly clear that there is a natural sense in which this is *cumulative*. In the first place \mathcal{S} is "contained in" $[\mathcal{S} \rightarrow \mathcal{S}]$ as a subspace: identify each $x \in \mathcal{S}$ with the corresponding constant function in $[\mathcal{S} \rightarrow \mathcal{S}]$. Clearly by our definitions this is an order-preserving correspondence. Also each $f \in [\mathcal{S} \rightarrow \mathcal{S}]$ is (crudely) approximated by a constant, namely $f(\mathbf{1})$ (this is the "best" element \sqsubseteq all the values $f(x)$). This relationship of subspace and approximation between *spaces* will be denoted by $\mathcal{S} \triangleleft [\mathcal{S} \rightarrow \mathcal{S}]$.

Pushing higher we can say

$$[\mathcal{S} \rightarrow \mathcal{S}] \triangleleft [[\mathcal{S} \rightarrow \mathcal{S}] \rightarrow \mathcal{S}],$$

but now for a *different* reason. Once we fix the reason why $\mathcal{S} \triangleleft [\mathcal{S} \rightarrow \mathcal{S}]$, we have to respect the function space structure of the higher \mathcal{F}_n . In the special case, suppose $f \in [\mathcal{S} \rightarrow \mathcal{S}]$. We want to inject f into the next space, so call it $i(f) \in [\mathcal{S} \rightarrow \mathcal{S}] \rightarrow \mathcal{S}$. If g is any element in $[\mathcal{S} \rightarrow \mathcal{S}]$ we are being required to define $i(f)(g) \in \mathcal{S}$. Now, since $g \in [\mathcal{S} \rightarrow \mathcal{S}]$, we have the original projec-

Fig. 3. The first chain of isomorphisms.

$$\begin{aligned} \mathcal{F}_n \times \mathcal{F}_n &= [\mathcal{F}_n \rightarrow \mathcal{S}] \times [\mathcal{F}_n \rightarrow \mathcal{S}] \\ &= [\mathcal{F}_n \rightarrow \mathcal{S} \times \mathcal{S}] \\ &= [\mathcal{F}_n \rightarrow \mathcal{S}] \\ &= \mathcal{F}_n \end{aligned}$$

Fig. 4. The second chain of isomorphisms.

$$\begin{aligned} [\mathcal{F}_n \rightarrow \mathcal{F}_n] &= [\mathcal{F}_n \rightarrow [\mathcal{F}_n \rightarrow \mathcal{S}]] \\ &= [[\mathcal{F}_n \times \mathcal{F}_n] \rightarrow \mathcal{S}] \\ &= [\mathcal{F}_n \rightarrow \mathcal{S}] \\ &= \mathcal{F}_n \end{aligned}$$

tion backwards $j(g) = g(\mathbf{1}) \in \mathcal{S}$. So, as this is the best approximation to g we can get in \mathcal{S} , we are stuck with defining

$$i(f)(g) = f(j(g)).$$

This gives the next map $i: \mathcal{F}_1 \rightarrow \mathcal{F}_2$. To define the corresponding projection $j: \mathcal{F}_2 \rightarrow \mathcal{F}_1$, we argue in a similar way and define

$$j(\phi)(x) = \phi(i(x)),$$

where we have $\phi \in [\mathcal{S} \rightarrow \mathcal{S}] \rightarrow \mathcal{S}$, and $i(x) \in [\mathcal{S} \rightarrow \mathcal{S}]$ is the constant function with value x . With this progression in mind there is no trouble in using an exactly similar plan in defining $i: \mathcal{F}_2 \rightarrow \mathcal{F}_3$ and $j: \mathcal{F}_3 \rightarrow \mathcal{F}_2$. And so on, giving the exact sense to the cumulation:

$$\mathcal{F}_0 \triangleleft \mathcal{F}_1 \triangleleft \mathcal{F}_2 \triangleleft \dots \triangleleft \mathcal{F}_n \triangleleft \mathcal{F}_{n+1} \triangleleft \dots$$

Having all this, it would be a pity not to pass to the limit (this time with *spaces*), and this is just what I want you to accept. What is obtained by decreeing that there is a space

$$\mathcal{F}_\infty = \lim_{n \rightarrow \infty} \mathcal{F}_n?$$

Since the separate stages interact thus:

$$\mathcal{F}_{n+1} = [\mathcal{F}_n \rightarrow \mathcal{S}],$$

it is not so queer to guess that

$$\mathcal{F}_\infty = [\mathcal{F}_\infty \rightarrow \mathcal{S}],$$

holds (at least up to isomorphism). It does, but I can only indicate the bare bones of the reason (and reasonableness) of this isomorphism. In the first place the separate spaces \mathcal{F}_n have been placed one inside another, which not only makes a tower of spaces but also respects the combination $f(x)$ as an algebraic operation of *two* variables. \mathcal{F}_∞ in a precise sense is the completion of the union of the \mathcal{F}_n ; that is *within* these spaces we can think of towers of *functions* each approximating the next (by the use of the i and j mappings), so that in \mathcal{F}_∞ these towers are given limits. If the towers are truncated, then we can argue that each space $\mathcal{F}_n \triangleleft \mathcal{F}_\infty$.

Now why the isomorphism on \mathcal{F}_∞ ? Take a function (continuous!) in $[\mathcal{F}_\infty \rightarrow \mathcal{S}]$. By its very continuity it will be determined by what it does to the *finite* levels \mathcal{F}_n .

That is, it will have better and better approximations in $[\mathcal{F}_n \rightarrow \mathcal{S}] = \mathcal{F}_{n+1}$; thus, the approximations "live" in the finite levels of \mathcal{F}_∞ . Their limit ought to just give us back the function $[\mathcal{F}_\infty \rightarrow \mathcal{S}]$ we started with. In the same way *any* element in \mathcal{F}_∞ can be regarded as a limit of approximate functions in the spaces $[\mathcal{F}_n \rightarrow \mathcal{S}]$. Admittedly there are details to check; but, in the limit, there is no real difference between \mathcal{F}_∞ and $[\mathcal{F}_\infty \rightarrow \mathcal{S}]$: the infinite level of higher type functions is its *own* function space. (As always: this is a consequence of continuity.)

Much structure is lurking under the surface here; in fact more than I thought at first. In Figure 3, I illustrate a chain of isomorphisms that shows that \mathcal{F} gets much of the character of \mathcal{S} with which we were already familiar. The reasons why these are valid are as follows. First, we treat \mathcal{F}_∞ as a function space. Now *pairs of functions* can be isomorphically put into correspondence with functions taking on *pairs of values*. But $\mathcal{S} \times \mathcal{S} = \mathcal{S}$ as we already know. The final step just puts functions on \mathcal{F}_∞ back to elements of \mathcal{F}_∞ .

Using the isomorphism of Figure 3, we can gain the further result illustrated in Figure 4. The reasons are fairly clear. Take a function from \mathcal{F}_∞ to \mathcal{F}_∞ . The values of this function can be construed as functions. But consider that a function whose values are functions is just (up to isomorphism of spaces) a *function of two arguments*. As we have seen in Figure 3, $\mathcal{F}_\infty \times \mathcal{F}_\infty = \mathcal{F}_\infty$, so we obtain the final simplification (up to isomorphism).

What we have done is to sketch why \mathcal{F}_∞ , the space of functions of infinite type, is a model of the λ -calculus. The λ -calculus is a language (*not* illustrated here), where every term can be regarded as denoting *both* an argument (or value) *and* a function at the same time. The *formal* details are pretty simple, but the *semantical* details are what we have been looking at: every element of the space \mathcal{F}_∞ can be taken at the same time as being an element of the space $[\mathcal{F}_\infty \rightarrow \mathcal{F}_\infty]$; thus, \mathcal{F}_∞ provides a model, but it is just one of many.

Without being able to be explicit, a denotational (or mathematical) semantics was outlined for a pure language of procedures (also *pairs* and all the other stuff in Figure 2). In the references cited on real programming languages, all the other features (of assignments, sequencing, declarations, etc., etc.) are added. What has been established in these references is that the method of semantical definition does in fact work. I hope you will look into it.

References

1. Böhm, C., Ed. *λ -Calculus and Computer Science Theory. Lecture Notes in Computer Science*, Vol. 37. Springer-Verlag, New York, 1975.
2. Clark, K.L., and Cowell, D.F. *Programs, Machines, and Computation*. McGraw-Hill, New York, 1976.
3. Crossley, J.N., ed. *Algebra and Logic*. Papers from the 1974 Summer Res. Inst. Australian Math. Soc., Monash U., Clayton, Victoria, Australia. *Lecture Notes in Mathematics*, Vol. 450, Springer-Verlag, New York, 1975.
4. Donahue, J.E. *Complementary Definitions of Programming Language Semantics. Lecture Notes in Computer Science*, Vol. 42, Springer-Verlag, 1976.
5. Eilenberg, S. *Automata, Languages, and Machines*. Academic Press, New York, 1974.
6. van Emden, M.H., and Kowalski, R.A. The semantics of predicate logic as a programming language. *J. ACM* 23, 4 (Oct. 1976), 733-742.
7. Manes, E.G., Ed. *Category Theory Applied to Computation and Control*. First Int. Symp. *Lecture Notes in Computer Science*, Vol. 25, Springer-Verlag, New York, 1976.
8. Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.
9. Milne, R., and Strachey, C. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and Wiley, New York, 2 Vols., 1976.
10. Plotkin, G.D. A powerdomain construction. *SIAM J. Comput.* 5 (1976), 452-487.
11. Rabin, M.O., and Scott, D.S. Finite automata and their decision problems. *IBM J. Res. and Develop.* 3 (1959), 114-125.
12. Scott, D.S. Data types as lattices. *SIAM J. Comput.* 5 (1976), 522-587.
13. Stoy, J.E. *Denotational Semantics - The Scott-Strachey Approach to Programming Language Theory*. M.I.T. Press, Cambridge, Mass. To appear.
14. Tennent, R.D. The denotational semantics of programming languages. *Comm. ACM* 19, 8 (Aug. 1976), 437-453.

1970 ACM Turing Lecture

Form and Content in Computer Science

MARVIN MINSKY

Massachusetts Institute of Technology, Cambridge, Massachusetts*

ABSTRACT. An excessive preoccupation with formalism is impeding the development of computer science. Form-content confusion is discussed relative to three areas: theory of computation, programming languages, and education.

KEY WORDS AND PHRASES: education, programming languages, compilers, theory of programming, heuristics, primary education, computer science curriculum, self-extending languages, "new mathematics"

CR CATEGORIES: 1.50, 3.66, 4.12, 4.29, 5.24

The trouble with computer science today is an obsessive concern with form instead of content.

No, that is the wrong way to begin. By any previous standard the vitality of computer science is enormous; what other intellectual area ever advanced so far in twenty years? Besides, the theory of computation perhaps encloses, in some way, the science of form, so that the concern is not so badly misplaced. Still, I will argue that an excessive preoccupation with formalism is impeding our development.

Before entering the discussion proper, I want to record the satisfaction my colleagues, students, and I derive from this Turing award. The cluster of questions, once philosophical but now scientific, surrounding the understanding of intelligence was of paramount concern to Alan Turing, and he along with a few other thinkers—notably Warren S. McCulloch and his young associate, Walter Pitts—made many of the early analyses that led both to the computer itself and to the new technology of artificial intelligence. In recognizing this area, this award should focus attention on other work of my own scientific family—especially Ray Solomonoff, Oliver Selfridge, John McCarthy, Allen Newell, Herbert Simon, and Seymour Papert, my closest associates in a decade of work. Papert's views pervade this essay.

This essay has three parts, suggesting form-content confusion in *theory of computation*, in *programming languages*, and in *education*.

1. *Theory of Computation*

To build a theory, one needs to know a lot about the basic phenomena of the subject matter. We simply do not know enough about these, in the theory of computation, to teach the subject very abstractly. Instead, we ought to teach more about the particular examples we now understand thoroughly, and hope that from this

* Project MAC and Electrical Engineering Department

we will be able to guess and prove more general principles. I am not saying this just to be conservative about things probably true that haven't been proved yet. I think that many of our beliefs that seem to be common sense are false. We have bad misconceptions about the possible exchanges between time and memory, trade-offs between time and program complexity, software and hardware, digital and analog circuits, serial and parallel computations, associative and addressed memory, and so on.

It is instructive to consider the analogy with physics, in which one can organize much of the basic knowledge as a collection of rather compact conservation laws. This, of course, is just one kind of description; one could use differential equations, minimum principles, equilibrium laws, etc. Conservation of energy, for example, can be interpreted as defining exchanges between various forms of potential and kinetic energies, such as between height and velocity squared, or between temperature and pressure-volume. One can base a development of quantum theory on trade-off between certainties of position and momentum, or between time and energy. There is nothing extraordinary about this; any equation with reasonably smooth solutions can be regarded as defining some kind of trade-off among its variable quantities. But there are many ways to formulate things and it is risky to become too attached to one particular form or law and come to believe that it is the *real* basic principle. See Feynman's [1] dissertation on this.

Nonetheless, the recognition of exchanges is often the conception of a science, if quantifying them is its birth. What do we have, in the computation field, of this character? In the theory of recursive functions, we have the observation by Shannon [2] that any Turing machine with Q states and R symbols is equivalent to one with 2 states and nQR symbols, and to one with 2 symbols and $n'QR$ states, where n and n' are small numbers. Thus the state-symbol product QR has an almost invariant quality in classifying machines. Unfortunately, one cannot identify the product with a useful measure of machine complexity because this, in turn, has a trade-off with the complexity of the encoding process for the machines—and that trade-off seems too inscrutable for useful application.

Let us consider a more elementary, but still puzzling, trade-off, that between addition and multiplication. How many multiplications does it take to evaluate the 3×3 determinant? If we write out the expansion as six triple-products, we need twelve multiplications. If we collect factors, using the distributive law, this reduces to nine. What is the minimum number, and how does one prove it, in this and in the $n \times n$ case? The important point is not that we need the answer. It is that we do not know how to tell or prove that proposed answers are correct! For a particular formula, one could perhaps use some sort of exhaustive search, but that wouldn't establish a general rule. One of our prime research goals should be to develop methods to prove that particular procedures are computationally minimal, in various senses.

A startling discovery was made about multiplication itself in the thesis of Cook [3], which uses a result of Toom; it is discussed in Knuth [4]. Consider the ordinary algorithm for multiplying decimal numbers: for two n -digit numbers this employs n^2 one-digit products. It is usually supposed that this is minimal. But suppose we write the numbers in two halves, so that the product is $N = (@A + B)(@C + D)$, where @ stands for multiplying by $10^{n/2}$. (The left-shift operation is considered to have negligible cost.) Then one can verify that

$$N = @@AC + BD + @((A + B)(C + D)) - @((AC + BD)).$$

This involves only *three* half-length multiplications, instead of the four that one might suppose were needed. For large n , the reduction can obviously be reapplied over and over to the smaller numbers. The price is a growing number of additions. By compounding this and other ideas, Cook showed that for any ϵ and large enough n , multiplication requires less than $n^{1+\epsilon}$ products, instead of the expected n^2 . Similarly, V. Strassen showed recently that to multiply two $m \times m$ matrices, the number of products could be reduced to the order of $m^{\log_2 7}$, when it was always believed that the number must be cubic—because there are m^2 terms in the result and each would seem to need a separate inner product with m multiplications. In both cases ordinary intuition has been wrong for a long time, so wrong that apparently no one looked for better methods. We still do not have a set of proof methods adequate for establishing exactly what is the minimum trade-off exchange, in the matrix case, between multiplying and adding.

The multiply-add exchange may not seem vitally important in itself, but if we cannot thoroughly understand something so simple, we can expect serious trouble with anything more complicated.

Consider another trade-off, that between memory size and computation time. In our book [5], Papert and I have posed a simple question: given an arbitrary collection of n -bit words, how many references to memory are required to tell which of those words is nearest¹ (in number of bits that agree) to an arbitrary given word? Since there are many ways to encode the “library” collection, some using more memory than others, the question stated more precisely is: how must the memory size grow to achieve a given reduction in the number of memory references? This much is trivial: if memory is large enough, only one reference is required, for we can use the question itself as address, and store the answer in the register so addressed. But if the memory is just large enough to store the information in the library, then one has to search all of it—and we do not know any intermediate results of any value. It is surely a fundamental theoretical problem of information retrieval, yet no one seems to have any idea about how to set a good lower bound on this basic trade-off.

Another is the serial-parallel exchange. Suppose that we had n computers instead of just one. How much can we speed up what kinds of calculations? For some, we can surely gain a factor of n . But these are rare. For others, we can gain $\log n$, but it is hard to find any or to prove what are their properties. And for most, I think, we can gain hardly anything; this is the case in which there are many highly branched conditionals, so that look-ahead on possible branches will usually be wasted. We know almost nothing about this; most people think, with surely incorrect optimism, that parallelism is usually a profitable way to speed up most computations.

These are just a few of the poorly understood questions about computational trade-offs. There is no space to discuss others, such as the digital-analog question. (Some problems about local versus global computations are outlined in [5].) And we know very little about trades between numerical and symbolic calculations.

There is, in today's computer science curricula, very little attention to what is known about such questions; almost all their time is devoted to formal classifications of syntactic language types, defeatist unsolvability theories, folklore about

¹ For identifying an *exact* match, one can use hash-coding and the problem is reasonably well understood.

systems programming, and generally trivial fragments of "optimization of logic design"—the latter often in situations where the art of heuristic programming has far outreached the special-case "theories" so grimly taught and tested—and invocations about programming style almost sure to be outmoded before the student graduates. Even the most seemingly abstract courses on recursive function theory and formal logic seem to ignore the few known useful results on proving facts about compilers or equivalence of programs. Most courses treat the results of work in artificial intelligence, some now fifteen years old, as a peripheral collection of special applications, whereas they in fact represent one of the largest bodies of empirical and theoretical exploration of real computational questions. Until all this preoccupation with form is replaced by attention to the substantial issues in computation, a young student might be well advised to avoid much of the computer science curricula, learn to program, acquire as much mathematics and other science as he can, and study the current literature in artificial intelligence, complexity, and optimization theories.

2. *Programming Languages*

Even in the field of programming languages and compilers, there is too much concern with form. I say "even" because one might feel that this is one area in which form *ought* to be the chief concern. But let us consider two assertions: (1) languages are getting so they have too much syntax, and (2) languages are being described with too much syntax.

Compilers are not concerned enough with the meanings of expressions, assertions, and descriptions. The use of context-free grammars for describing fragments of languages led to important advances in uniformity, both in specification and in implementation. But although this works well in simple cases, attempts to use it may be retarding development in more complicated areas. There are serious problems in using grammars to describe self-modifying or self-extending languages that involve executing, as well as specifying, processes. One cannot describe syntactically—that is, statically—the valid expressions of a language that is changing. Syntax extension mechanisms must be described, to be sure, but if these are given in terms of a modern pattern-matching language such as SNOBOL, CONVERT [6], or MATCHLESS [7], there need be no distinction between the parsing program and the language description itself. Computer languages of the future will be more concerned with goals and less with procedures specified by the programmer. The following arguments are a little on the extreme side but, in view of today's preoccupation with form, this overstepping will do no harm. (Some of the ideas are due to C. Hewitt and T. Winograd.)

2.1. **SYNTAX IS OFTEN UNNECESSARY.** One can survive with much less syntax than is generally realized. Much of programming syntax is concerned with suppression of parentheses or with emphasis of scope markers. There are alternatives that have been much underused.

Please do not think that I am against the use, at the human interface, of such devices as infixes and operator precedence. They have their place. But their importance to computer science as a whole has been so exaggerated that it is beginning to corrupt the youth.

Consider the familiar algorithm for the square root, as it might be written in a

modern algebraic language, ignoring such matters as declarations of data types. One asks for the square root of A, given an initial estimate X and an error limit E.

DEFINE SQRT(A,X,E):
 if ABS(A - X * X) < E then X else SQRT(A, (X + A + X) / 2, E).

In an imaginary but recognizable version of LISP (see Levin [8] or Weissman [9]), this same procedure might be written:

```
(DEFINE (SQRT A X E)
  (IF (LESS (ABS (- A (* X X))) E) THEN X
    ELSE (SQRT A (/ (+ X (/ A X)) 2) E)))
```

Here, the function names come immediately *inside* their parentheses. The clumsiness, for humans, of writing all the parentheses is evident; the advantages of not having to learn all the conventions, such as that $(X + A \div X)$ is $(+ X (\div A X))$ and not $(\div (+ X A) X)$, is often overlooked.

It remains to be seen whether a syntax with explicit delimiters is reactionary, or whether it is the wave of the future. It has important advantages for editing, interpreting, and for *creation of programs by other programs*. The complete syntax of LISP can be learned in an hour or so; the interpreter is compact and not exceedingly complicated, and students often can answer questions about the system by reading the interpreter program itself. Of course, this will not answer *all* questions about a real, practical implementation, but neither would any feasible set of syntax rules. Furthermore, despite the language's clumsiness, many frontier workers consider it to have outstanding expressive power. Nearly all work on procedures that solve problems by building and modifying hypotheses have been written in this or related languages. Unfortunately, language designers are generally unfamiliar with this area, and tend to dismiss it as a specialized body of "symbol-manipulation techniques."

Much can be done to clarify the structure of expressions in such a "syntax-weak" language by using indentation and other layout devices that are outside the language proper. For example, one can use a "postponement" symbol that belongs to an input preprocessor to rewrite the above as

```
DEFINE (SQRT A X E) ↓ .
  IF ↓ THEN X ELSE ↓ .
    LESS (ABS ↓ ) E.
      - A (* X X).
    SQRT A ↓ E.
      + ↓ 2.
      + X (/ A X)
```

where the dot means ")" and the arrow means "insert here the next expression, delimited by a dot, that is available after replacing (recursively) its own arrows." The indentations are optional. This gets a good part of the effect of the usual scope indicators and conventions by two simple devices, both handled trivially by reading programs, and it is easy to edit because subexpressions are usually complete on each line.

To appreciate the power and limitations of the postponement operator, the reader should take his favorite language and his favorite algorithms and see what happens. He will find many choices of what to postpone, and he exercises judgment about what to say first, which arguments to emphasize, and so forth. Of course, ↓ is not

the answer to all problems; one needs a postponement device also for list fragments, and that requires its own delimiter. In any case, these are but steps toward more graphical program-description systems, for we will not forever stay confined to mere strings of symbols.

Another expository device, suggested by Dana Scott, is to have alternative brackets for indicating right-to-left functional composition, so that one can write $\langle\langle x \rangle h \rangle g \circ f$ instead of $f(g(h(x)))$ when one wants to indicate more naturally what happens to a quantity in the course of a computation. This allows different "accents," as in $f(\langle h(x) \rangle g)$, which can be read: "Compute f of what you get by first computing $h(x)$ and then applying g to it."

The point is better made, perhaps, by analogy than by example. In their fanatic concern with syntax, language designers have become too sentence oriented. With such devices as \Downarrow , one can construct objects that are more like paragraphs, without falling all the way back to flow diagrams.

Today's high level programming languages offer little expressive power in the sense of flexibility of style. One cannot control the sequence of presentation of ideas very much without changing the algorithm itself.

2.2. EFFICIENCY AND UNDERSTANDING PROGRAMS. What is a compiler for? The usual answers resemble "to translate from one language to another" or "to take a description of an algorithm and assemble it into a program, filling in many small details." For the future, a more ambitious view is required. Most compilers will be systems that "produce an algorithm, given a description of its effect." This is already the case for modern picture-format systems; they do all the creative work, while the user merely supplies examples of the desired formats: here the compilers are more expert than the users. Pattern-matching languages are also good examples. But except for a few such special cases, the compiler designers have made little progress in getting good programs written. Recognition of common subexpressions, optimization of inner loops, allocation of multiple registers, and so forth, lead but to small linear improvements in efficiency—and compilers do little enough about even these. Automatic storage assignments can be worth more. But the real payoff is in analysis of the *computational content* of the algorithm itself, rather than the way the programmer wrote it down. Consider, for example:

```
DEFINE FIB(N): if N=1 then 1, if N=2 then 1,
else FIB(N-1) + FIB(N-2).
```

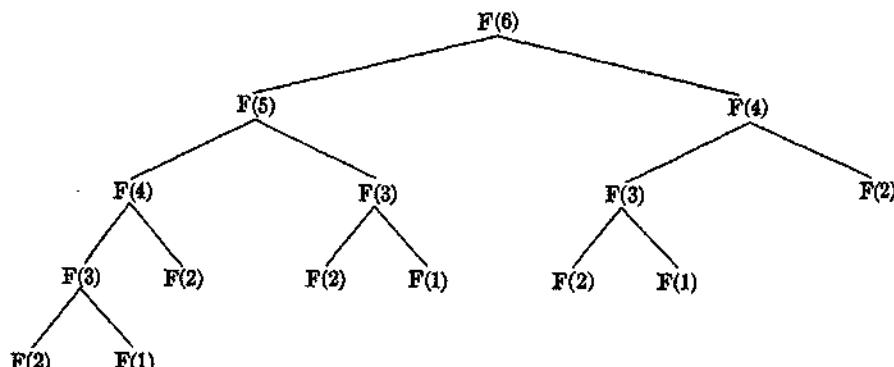


FIG. 1

This recursive definition of the Fibonacci numbers 1, 1, 2, 3, 5, 8, 13, ... can be given to any respectable algorithmic language and will result in the branching tree of evaluation steps shown in Figure 1.

One sees that the amount of work the machine will do grows exponentially with N . (More precisely, it passes through the order of $\text{FIB}(N)$ evaluations of the definition!) There are better ways to compute this function. Thus we can define two temporary registers and evaluate $\text{FIB}(N 1 1)$ in

DEFINE $\text{FIB}(N A B)$: if $N=1$ then A else $\text{FIB}(N-1 A+B A)$.

which is singly recursive and avoids the branching tree, or even use

```

A = 0
B = 1
LOOP SWAP A B
if N=1 return A
N = N-1
B = A+B
goto LOOP

```

Any programmer will soon think of these, once he sees what happens in the branching evaluation. This is a case in which a "course-of-values" recursion can be transformed into a simple iteration. Today's compilers don't recognize even simple cases of such transformations, although the reduction in exponential order outweighs any possible gains in local "optimization" of code. It is no use protesting either that such gains are rare or that such matters are the programmer's responsibility. If it is important to save compiling time, then such abilities could be excised. For programs written in the pattern-matching languages, for example, such simplifications are indeed often made. One usually wins by compiling an efficient tree-parser for a BNF system instead of executing brute force analysis-by-synthesis.

To be sure, a systematic theory of such transformations is difficult. A system will have to be pretty smart to detect which transformations are relevant and when it pays to use them. Since the programmer already knows his intent, the problem would often be easier if the proposed algorithm is accompanied (or even replaced) by a suitable goal-declaration expression.

To move in this direction, we need a body of knowledge about analyzing and synthesizing programs. On the theoretical side there is now a lot of activity studying the equivalence of algorithms and schemata, and on proving that procedures have stated properties. On the practical side the works of W. A. Martin [10] and J. Moses [11] illustrate how to make systems that know enough about symbolic transformations of particular mathematical techniques to significantly supplement the applied mathematical abilities of their users.

There is no practical consequence to the fact that the program-reduction problem is recursively unsolvable, in general. In any case one would expect programs eventually to go far beyond human ability in this activity, and make use of a large body of program transformations in formally purified forms. These will not be easy to apply directly. Instead, one can expect the development to follow the lines we have seen in symbolic integration, e.g. as in Slagle [12] and Moses [11]. First a set of simple formal transformations that correspond to the elementary entries of a Table

of Integrals was developed. On top of these Slagle built a set of heuristic techniques for the algebraic and analytic transformation of a practical problem into those already understood elements; this involved a set of characterization and matching procedures that might be said to use "pattern recognition." In the system of Moses both the matching procedures and the transformations were so refined that, in most practical problems, the heuristic search strategy that played a large part in the performance of Slagle's program became a minor augmentation of the sure knowledge and its skillful application comprised in Moses' system. A heuristic compiler system will eventually need much more *general knowledge* and common sense than did the symbolic integration systems, for its goal is more like making a whole mathematician than a specialized integrator.

2.3. DESCRIBING PROGRAMMING SYSTEMS. No matter how a language is described, a computer must use a procedure to interpret it. One should remember that *in describing a language the main goal is to explain how to write programs in it and what such programs mean*. The main goal isn't to describe the syntax.

Within the static framework of syntax rules, normal forms, Post productions, and other such schemes, one obtains the equivalents of logical systems with axioms, rules of inference, and theorems. To design an unambiguous syntax corresponds then to designing a mathematical system in which each theorem has exactly one proof! But in the computational framework, this is quite beside the point. One has an extra ingredient—control—that lies outside the usual framework of a logical system; an additional set of rules that specify when a rule of inference is to be used. So, for many purposes, ambiguity is a pseudoproblem. If we view a program as a process, we can remember that our most powerful process-describing tools are programs themselves, and *they* are inherently unambiguous.

There is no paradox in defining a programming language by a program. The procedural definition must be understood, of course. One can achieve this understanding by definitions written in another language, one that may be different, more familiar, or simpler than the one being defined. But it is often practical, convenient, and proper to use the same language! For to understand the definition, one needs to know only the working of that particular program, and not all implications of all possible applications of the language. It is this particularization that makes bootstrapping possible, a point that often puzzles beginners as well as apparent authorities.

Using BNF to describe the formation of expressions may be retarding development of new languages that smoothly incorporate quotation, self-modification, and symbolic manipulation into a traditional algorithmic framework. This, in turn, retards progress toward problem-solving, goal-oriented programming systems. Paradoxically, though modern programming ideas were developed because processes were hard to depict with classical mathematical notations, designers are turning back to an earlier form—the equation—in just the kind of situation that *needs* program. In Section 3, which is on education, a similar situation is seen in teaching, with perhaps more serious consequences.

3. *Learning, Teaching, and the "New Mathematics"*

Education is another area in which the computer scientist has confused form and content, but this time the confusion concerns his professional role. He perceives his

principal function to provide programs and machines for use in old and new educational schemes. Well and good, but I believe he has a more complex responsibility—to work out and communicate models of the process of education itself.

In the discussion below, I sketch briefly the viewpoint (developed with Seymour Papert) from which this belief stems. The following statements are typical of our view:

—To help people learn is to help them build, in their heads, various kinds of computational models.

—This can best be done by a teacher who has, in his head, a reasonable model of what is in the pupil's head.

—For the same reason the student, when debugging his own models and procedures, should have a model of what he is doing, and must know good debugging techniques, such as how to formulate simple but critical test cases.

—It will help the student to know something about computational models and programming. The idea of debugging² itself, for example, is a very powerful concept—in contrast to the helplessness promoted by our cultural heritage about gifts, talents, and aptitudes. The latter encourages "I'm not good at this" instead of "How can I make myself better at it?"

These have the sound of common sense, yet they are not among the basic principles of any of the popular educational schemes such as "operant reinforcement," "discovery methods," audio-visual synergism, etc. This is not because educators have ignored the possibility of mental models, but because they simply had no effective way, before the beginning of work on simulation of thought processes, to describe, construct, and test such ideas.

We cannot digress here to answer skeptics who feel it too simpleminded (if not impious, or obscene) to compare minds with programs. We can refer many such critics to Turing's paper [13]. For those who feel that the answer cannot lie in any machine, digital or otherwise, one can argue [14] that machines, when they become intelligent, very likely will feel the same way. For some overviews of this area, see Feigenbaum and Feldman [15] and Minsky [16]; one can keep really up-to-date in this fast-moving field only by reading the contemporary doctoral theses and conference papers on artificial intelligence.

There is a fundamental pragmatic point in favor of our propositions. The child needs models: to understand the city he may use the organism model; it must eat, breathe, excrete, defend itself, etc. Not a very good model, but useful enough. The metabolism of a real organism he can understand, in turn, by comparison with an engine. But to model his own self he *cannot* use the engine or the organism or the city or the telephone switchboard; nothing will serve at all but the computer with its programs and their bugs. Eventually, programming itself will become more important even than mathematics in early education. Nevertheless I have chosen *mathematics* as the subject of the remainder of this paper, partly because we understand it better but mainly because the prejudice against programming as an academic subject would provoke too much resistance. Any other subject could also do, I suppose, but mathematical issues and concepts are the sharpest and least confused by highly charged emotional problems.

²Turing was quite good at debugging hardware. He would leave the power on, so as not to lose the "feel" of the thing. Everyone does that today, but it is not the same thing now that the circuits all work on three or five volts.

3.1. MATHEMATICAL PORTRAIT OF A SMALL CHILD. Imagine a small child of between five and six years, about to enter the first grade. If we extrapolate today's trend, his mathematical education will be conducted by poorly oriented teachers and, partly, by poorly programmed machines; neither will be able to respond to much beyond "correct" and "wrong" answers, let alone to make reasonable interpretations of what the child does or says, because neither will contain good models of the children, or good theories of children's intellectual development. The child will begin with simple arithmetic, set theory, and a little geometry; ten years later he will know a little about the formal theory of the real numbers, a little about linear equations, a little more about geometry, and almost nothing about continuous and limiting processes. He will be an adolescent with little taste for analytical thinking, unable to apply the ten years' experience to understanding his new world.

Let us look more closely at our young child, in a composite picture drawn from the work of Piaget and other observers of the child's mental construction.

Our child will be able to say "one, two, three, . . ." at least up to thirty and probably up to a thousand. He will know the names of some larger numbers but will not be able to see, for example, why ten thousand is a hundred hundred. He will have serious difficulty in counting backwards unless he has recently become very interested in this. (Being good at it would make simple subtraction easier, and might be worth some practice.) He doesn't have much feeling for odd and even.

He can count four to six objects with perfect reliability, but he will not get the same count every time with fifteen scattered objects. He will be annoyed with this, because he is quite sure he should get the same number each time. The observer will therefore think the child has a good idea of the number concept but that he is not too skillful at applying it.

However, important aspects of his concept of number will not be at all secure by adult standards. For example, when the objects are rearranged before his eyes, his impression of their quantity will be affected by the geometric arrangement. Thus he will say that there are fewer x 's than y 's in:

$$\begin{array}{ccccccccc} x & x & x & x & x & x & x \\ y & y & y & y & y & y & y \end{array}$$

and when we move the x 's to

$$\begin{array}{ccccccccc} x & x & x & x & x & x & x \\ y & y & y & y & y & y & y \end{array}$$

he will say there are more x 's than y 's. To be sure, he is answering (in his own mind) a different question about size, quite correctly, but this is exactly the point: the immutability of the number, in such situations, has little grip on him. He cannot use it effectively for reasoning although he shows, on questioning, that he knows that the number of things cannot change simply because they are rearranged. Similarly, when water is poured from one glass to another (Figure 2(a)), he will say that there is more water in the tall jar than in the squat one. He will have poor estimates about plane areas, so that we will not be able to find a context in which he treats the larger area in Figure 2(b) as four times the size of the smaller one. When he is an adult, by the way, and is given two vessels, one twice as large as the other, in all dimensions (Figure 2(c)), he will think the one holds about four times as much as the other: probably he will never acquire better estimates of volume.

As for the numbers themselves, we know little of what is in his mind. According to Galton [17], thirty children in a hundred will associate small numbers with defi-

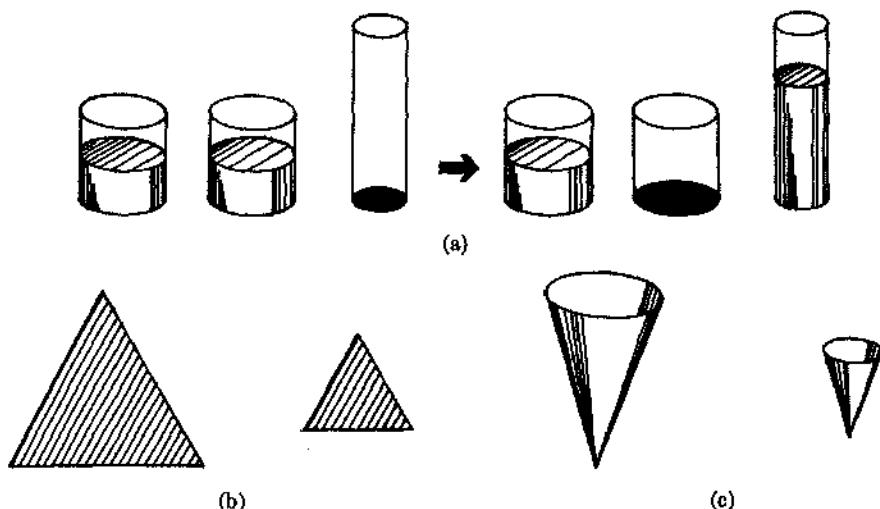


FIG. 2

nite visual locations in the space in front of their body image, arranged in some idiosyncratic manner such as that shown in Figure 3. They will probably still retain these as adults, and may use them in some obscure semiconscious way to remember telephone numbers; they will probably grow different spatio-visual representations for historical dates, etc. The teachers will never have heard of such a thing and, if a child speaks of it, even the teacher with her own number form is unlikely to respond with recognition. (My experience is that it takes a series of carefully posed questions before one of these adults will respond, "Oh, yes; 3 is over there, a little farther back.") When our child learns column sums, he may keep track of carries by setting his tongue to certain teeth, or use some other obscure device for temporary memory, and no one will ever know. Perhaps some ways are better than others.

His geometric world is different from ours. He does not see clearly that triangles are rigid, and thus different from other polygons. He does not know that a 100-line approximation to a circle is indistinguishable from a circle unless it is quite large. He does not draw a cube in perspective. He has only recently realized that squares become diamonds when put on their points. The perceptual distinction persists in adults. Thus in Figure 4 we see, as noted by Attneave [18], that the impression of square versus diamond is affected by other alignments in the scene, evidently by

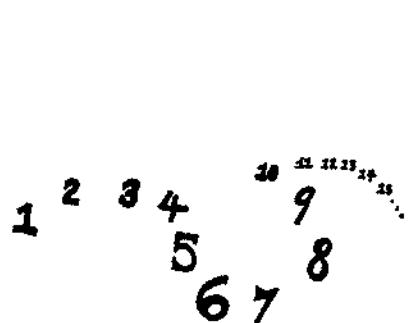


FIG. 3

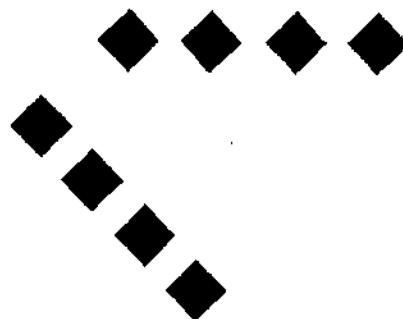


FIG. 4

determining our choice of which axis of symmetry is to be used in the subjective description.

Our child understands the topological idea of enclosure quite well. Why? This is a very complicated concept in classical mathematics but in terms of computational processes it is perhaps not so difficult. But our child is almost sure to be muddled about the situation in Figure 5 (see Papert [19]): "When the bus begins its trip around the lake, a boy is seated on the side away from the water. Will he be on the lake side at some time in the trip?" Difficulty with this is liable to persist through the child's eighth year, and perhaps relates to his difficulties with other abstract double reversals such as in subtracting negative numbers, or with apprehending other consequences of continuity—"At what point in the trip is there any sudden change?"—or with other bridges between local and global.

Our portrait is drawn in more detail in the literature on developmental psychology. But no one has yet built enough of a computational model of a child to see how these abilities and limitations link together in a structure compatible with (and perhaps consequential to) other things he can do so effectively. Such work is beginning, however, and I expect the next decade to see substantial progress on such models.

If we knew more about these matters, we might be able to help the child. At present we don't even have good diagnostics: his apparent ability to learn to give correct answers to formal questions may show only that he has developed some isolated library routines. If these cannot be called by his central problem-solving programs, because they use incompatible data structures or whatever, we may get a high rated test-passes who will never think very well.

Before computation, the community of ideas about the nature of thought was too feeble to support an effective theory of learning and development. Neither the finite-state models of the Behaviorists, the hydraulic and economic analogies of the Freudians, nor the rabbit-in-the-hat insights of the Gestaltists supplied enough ingredients to understand so intricate a subject. It needs a substrate of already debugged theories and solutions of related but simpler problems. Now we have a flood of such ideas, well defined and implemented, for thinking about thinking; only a fraction are represented in traditional psychology:

symbol table	closed subroutines
pure procedure	pushdown list
time-sharing	interrupt
calling sequence	communication cell
functional argument	common storage
memory protection	decision tree
dispatch table	hardware-software trade-off
error message	serial-parallel trade-off
function-call trace	time-memory trade-off
breakpoint	conditional breakpoint
languages	asynchronous processor
compiler	interpreter
indirect address	garbage collection
macro	list structure
property list	block structure
data type	look-ahead
hash coding	look-behind
microprogram	diagnostic program
format matching	executive program

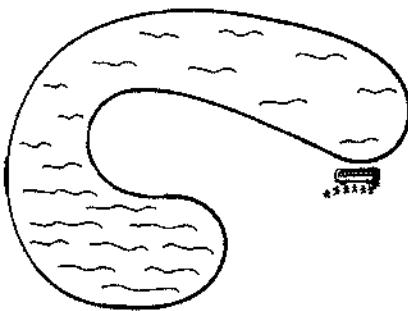


FIG. 5

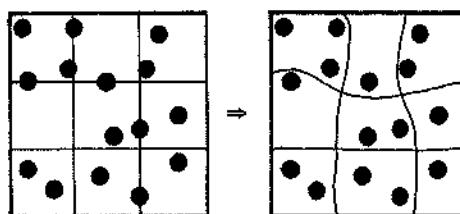


FIG. 6

These are just a few ideas from general systems programming and debugging; we have said nothing about the many more specifically relevant concepts in languages or in artificial intelligence or in computer hardware or other advanced areas. All these serve today as tools of a curious and intricate craft, programming. But just as astronomy succeeded astrology, following Kepler's regularities, the discovery of principles in empirical explorations of intellectual process in machines should lead to a science. (In education we face still the same competition! The *Boston Globe* has an astrology page in its "comics" section. Help fight intellect pollution!)

To return to our child, how can our computational ideas help him with his number concept? As a baby he learned to recognize certain special pair configurations such as two hands or two shoes. Much later he learned about some threes—perhaps the long gap is because the environment doesn't have many fixed triplets: if he happens to find three pennies he will likely lose or gain one soon. Eventually he will find some procedure that manages five or six things, and he will be less at the mercy of finding and losing. But for more than six or seven things, he will remain at the mercy of forgetting; even if his verbal count is flawless, his enumeration procedure will have defects. He will skip some items and count others twice. We can help by proposing better procedures; putting things into a box is nearly foolproof, and so is crossing them off. But for fixed objects he will need some mental grouping procedure.

First one should try to know what the child is doing; eye-motion study might help, asking him might be enough. He may be selecting the next item with some unreliable, nearly random method, with no good way to keep track of what has been counted. We might suggest: *sliding a cursor; inventing easily remembered groups; drawing a coarse mesh.*

In each case the construction can be either real or imaginary. In using the mesh method one has to remember not to count twice objects that cross the mesh lines. The teacher should show that it is good to plan ahead, as in Figure 6, distorting the mesh to avoid the ambiguities! Mathematically, the important concept is that "every proper counting procedure yields the same number." The child will understand that any algorithm is proper which (1) *counts all the objects*, (2) *counts none of them twice*.

Perhaps this procedural condition seems too simple; even an adult could understand it. In any case, it is not the concept of number adopted in what is today generally called the 'New Math,' and taught in our primary schools. The following polemic discusses this.

3.2. THE "NEW MATHEMATICS." By the "new math" I mean certain primary school attempts to imitate the formalistic outputs of professional mathematicians. Precipitously adopted by many schools in the wake of broad new concerns with early education, I think the approach is generally bad because of form-content displacements of several kinds. These cause problems for the teacher as well as for the child.

Because of the formalistic approach the teacher will not be able to help the child very much with problems of formulation. For she will feel insecure herself as she drills him on such matters as the difference between the empty set and nothing, or the distinction between the "numeral" $3+5$ and the numeral 8 which is the "common name" of the number eight, hoping that he will not ask what is the common name of the fraction $\frac{8}{1}$, which is probably different from the rational $\frac{8}{1}$ and different from the quotient $\frac{8}{1}$ and different from the "indicated division" $\frac{8}{1}$ and different from the ordered pair $(8, 1)$. She will be reticent about discussing parallel lines. For parallel lines do not usually meet, she knows, but they might (she has heard) if produced far enough, for did not something like that happen once in an experiment by some Russian mathematicians? But enough of the problems of the teacher: let us consider now three classes of objections from the child's standpoint.

Developmental Objections. It is very bad to insist that the child keep his knowledge in a simple ordered hierarchy. In order to retrieve what he needs, he must have a multiply connected network, so that he can try several ways to do each thing. He may not manage to match the first method to the needs of the problem. Emphasis on the "formal proof" is destructive at this stage, because the knowledge needed for finding proofs, and for understanding them, is far more complex (and less useful) than the knowledge mentioned *in proofs*. The network of knowledge one needs for understanding geometry is a web of examples and phenomena, and observations about the similarities and differences between them. One does not find evidence, in children, that such webs are ordered like the axioms and theorems of a logistic system, or that the child could use such a lattice if he had one. After one understands a phenomenon, it may be of great value to make a formal system for it, to make it easier to understand more advanced things. But even then, such a formal system is just one of many possible models; the New Math writers seem to confuse their axiom-theorem model with the number system itself. In the case of the axioms for arithmetic, I will now argue, the formalism is often likely to do more harm than good for the understanding of more advanced things.

Historically, the "set" approach used in New Math comes from a formalist attempt to derive the intuitive properties of the continuum from a nearly finite set theory. They partly succeeded in this stunt (or "hack," as some programmers would put it), but in a manner so complex that one cannot talk seriously about the real numbers until well into high school, if one follows this model. The ideas of topology are deferred until much later. But children in their sixth year already have well-developed geometric and topological ideas, only they have little ability to manipulate abstract symbols and definitions. We should build out from the child's strong points, instead of undermining him by attempting to replace what he has by structures he cannot yet handle. But it is just like mathematicians—certainly the world's worst expositors—to think: "You can teach a child anything, if you just get the definitions precise enough," or "If we get all the definitions right the first time, we won't have any trouble later." We are not programming an

empty machine in FORTRAN: we are meddling with a poorly understood large system that, characteristically, uses multiply defined symbols in its normal heuristic behavior.

Intuitive Objections. New Math emphasizes the idea that a number can be identified with an equivalence class of all sets that can be put into one-to-one correspondence with one another. Then the rational numbers are defined as equivalence classes of pairs of integers, and a maze of formalism is introduced to prevent the child from identifying the rationals with the quotients or fractions. Functions are often treated as sets, although some texts present "function machines" with a superficially algorithmic flavor. The definition of a "variable" is another fiendish maze of complication involving names, values, expressions, clauses, sentences, numerals, "indicated operations," and so forth. (In fact, there are so many different kinds of data in real problem-solving that real-life mathematicians do *not* usually give them formal distinctions, but use the entire problem context to explain them.) In the course of pursuing this formalistic obsession, the curriculum never presents any coherent picture of real mathematical phenomena of processes, discrete or continuous; of the algebra whose notational syntax concerns it so; or of geometry. The "theorems" that are "proved" from time to time, such as, "A number x has only one additive inverse, $-x$," are so mundane and obvious that neither teacher nor student can make out the purpose of the proof. The "official" proof would add y to both sides of $x + (-y) = 0$, apply the associative law, then the commutative law, then the $y + (-y) = 0$ law, and finally the axioms of equality, to show that y must equal x . The child's mind can more easily understand deeper ideas: "In $x + (-y) = 0$, if y were less than x there would be some left over; while if x were less than y there would be a minus number left—so they must be exactly equal." The child is not permitted to use this kind of order-plus-continuity thinking, presumably because it uses "more advanced knowledge," hence isn't part of a "real proof." But in the network of ideas the child needs, this link has equal logical status and surely greater heuristic value. For another example, the student is made to distinguish clearly between the *inverse* of addition and the *opposite* sense of distance, a discrimination that seems entirely against the fusion of these notions that would seem desirable.

Computational Objections. The idea of a procedure, and the know-how that comes from learning how to test, modify, and adapt procedures, can transfer to many of the child's other activities. Traditional academic subjects such as algebra and arithmetic have relatively small developmental significance, especially when they are weak in intuitive geometry. (The question of which kinds of learning can "transfer" to other activities is a fundamental one in educational theory: I emphasize again our conjecture that the ideas of procedures and debugging will turn out to be unique in their transferability.) In algebra, as we have noted, the concept of "variable" is complicated; but in computation the child can easily see " $x + y + z$ " as describing a procedure (any procedure for adding!) with " x ," " y ," and " z " as pointing to its "data." Functions are easy to grasp as procedures, hard if imagined as ordered pairs. If you want a graph, describe a machine that draws the graph; if you have a graph, describe a machine that can read it to find the values of the function. Both are easy and useful concepts.

Let us not fall into a cultural trap: the set theory "foundation" for mathematics is popular today among mathematicians because it is the one they tackled and mas-

tered (in college). These scientists simply are not acquainted, generally, with computation or with the Post-Turing-McCulloch-Pitts-McCarthy-Newell-Simon... family of theories that will be so much more important when the children grow up. Set theory is not, as the logicians and publishers would have it, *the only and true foundation of mathematics*; it is a viewpoint that is pretty good for investigating the transfinite, but undistinguished for comprehending the real numbers, and quite substandard for learning about arithmetic, algebra, and geometry.

To summarize my objections, the New Math emphasizes the use of formalism and symbolic manipulation instead of the heuristic and intuitive content of the subject matter. The child is expected to learn how to solve problems but we do not teach him what we know, either about the subject or about problem-solving.³

As an example of how the preoccupation with form (in this case, the axioms for arithmetic) can warp one's view of the content, let us examine the weird compulsion to insist that addition is ultimately an operation on just two quantities. In New Math, $a+b+c$ must "really" be one of $(a+(b+c))$ or $((a+b)+c)$, and $a+b+c+d$ can be meaningful only after several applications of the associative law. Now this is silly in many contexts. The child has already a good intuitive idea of what it means to put several sets together; it is just as easy to mix five colors of beads as two. Thus addition is already an n -ary operation. But listen to the book trying to prove that this is not so:

Addition is . . . always performed on two numbers. This may not seem reasonable at first sight, since you have often added long strings of figures. Try an experiment on yourself. Try to add the numbers 7, 8, 3 simultaneously. No matter how you attempt it, you are forced to choose two of the numbers, add them, and then add the third to their sum.

—From a ninth-grade text

Is the height of a tower the result of adding its stages by pairs in a certain order? Is the length or area of an object produced that way from its parts? Why did they introduce their sets and their one-one correspondences then to miss the point? Evidently, they have talked themselves into believing that the axioms they selected for algebra have some special kind of truth!

Let us consider a few important and pretty ideas that are not discussed much in grade school. First consider the sum $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$. Interpreted as area, one gets fascinating regrouping ideas, as in Figure 7. Once the child knows how to do division, he can compute and appreciate some quantitative aspects of the limiting process .5, .75, .875, .9375, .96875, ..., and he can learn about folding and cutting and epidemics and populations. He could learn about $x = px + qx$, where $p + q = 1$, and hence appreciate dilution; he can learn that $\frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{6}{7}, \frac{7}{8}, \dots \rightarrow 1$ and begin to understand the many colorful and common-sense geometrical and topological consequences of such matters.

But in the New Math, the syntactic distinctions between rationals, quotients, and fractions are carried so far that to see which of $\frac{5}{8}$ and $\frac{4}{6}$ is larger, one is not

³ In a shrewd but hilarious discussion of New Math textbooks, Feynman [20] explores the consequences of distinguishing between the thing and itself. "Color the picture of the ball red," a book says, instead of "Color the ball red." "Shall we color the entire square area in which the ball image appears or just the part inside the circle of the ball?" asks Feynman. (To "color the balls red" would presumably have to be "color the insides of the circles of all the members of the set of balls" or something like that.)

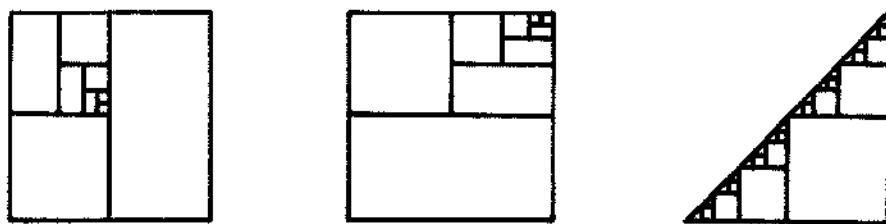


FIG. 7

permitted to compute and compare .375 with .4444. One must cross-multiply. Now cross-multiplication is very cute, but it has two bugs: (1) no one can remember which way the resulting conditional should branch, and (2) it doesn't tell how far apart the numbers are. The abstract concept of order is very elegant (another set of axioms for the obvious) but the children already understand order pretty well and want to know the amounts.

Another obsession is the concern for number base. It is good for the children to understand clearly that 223 is "two hundred" plus "twenty" plus "three," and I think that this should be made as simple as possible rather than complicated.⁴ I do not think that the idea is so rich that one should drill young children to do arithmetic in several bases! For there is very little transfer of this feeble concept to other things, and it risks a crippling insult to the fragile arithmetic of pupils who, already troubled with $6 + 7 = 13$, now find that $6 + 7 = 15$. Besides, for all the attention to number base, I do not see in my children's books any concern with even a few nontrivial implications—concepts that might justify the attention, such as:

Why is there only one way to write a decimal integer?

Why does casting out nines work? (It isn't even mentioned.)

What happens if we use arbitrary nonpowers, such as $a + 37b + 24c + 11d + \dots$ instead of the usual $a + 10b + 100c + 1000d + \dots$?

If they don't discuss such matters, they must have another purpose. My conjecture is that the whole fuss is to make the kids better understand the procedures for multiplying and dividing. But from a developmental viewpoint this may be a serious mistake—in the strategies of both the old and the "new" mathematical curricula. At best, the standard algorithm for long division is cumbersome, and most children will never use it to explore numeric phenomena. And, although it is of some interest to understand how it works, writing out the whole display suggests that the educator believes that the child ought to understand the horrible thing every time! This is wrong. The important idea, if any, is the repeated subtraction; the rest is just a clever but not vital programming hack.

If we can teach, perhaps by rote, a practical division algorithm, fine. But in any case let us give them little calculators; if that is too expensive, why not slide rules. Please, without an impossible explanation. The important thing is to get on to the real numbers! The New Math's concern with integers is so fanatical that it reminds me, if I may mention another pseudoscience, of numerology. (How about that, *Boston Globe!*)

The Cauchy-Dedekind-Russell-Whitehead set-theory formalism was a large accomplishment—another (following Euclid) of a series of demonstrations that many

⁴ Cf. Tom Lehrer's song, "New Math" [21].

mathematical ideas can be derived from a few primitives, albeit by a long and tortuous route. But the child's problem is to acquire the ideas at all; he needs to learn about reality. In terms of the concepts available to him, the entire formalism of set theory cannot hold a candle to one older, simpler, and possibly greater idea: the nonterminating decimal representation of the intuitive real number line.

There is a real conflict between the logician's goal and the educator's. The logician wants to minimize the variety of ideas, and doesn't mind a long, thin path. The educator (rightly) wants to make the paths short and doesn't mind—in fact, prefers—connections to many other ideas. And he cares almost not at all about the directions of the links.

As for better understanding of the integers, countless exercises in making little children draw diagrams of one-one correspondences will not help, I think. It will help, no doubt, in their learning valuable algorithms, not for number but for the important topological and procedural problems in drawing paths without crossing, and so forth. It is just that sort of problem, now treated entirely accidentally, that we should attend to.

The computer scientist thus has a responsibility to education. Not, as he thinks, because he will have to program the teaching machines. Certainly not because he is a skilled user of "finite mathematics." He knows how to debug programs; he must tell the educators how to help the children to debug their own problem-solving processes. He knows how procedures depend on their data structures; he can tell educators how to prepare children for new ideas. He knows why it is bad to use double-purpose tricks that haunt one later in debugging and enlarging programs. (Thus, one can capture the kids' interest by associating small numbers with arbitrary colors. But what will this trick do for their later attempts to apply number ideas to area, or to volume, or to value?) The computer scientist is the one who must study such matters, because he is the proprietor of the concept of procedure, the secret educators have so long been seeking.

REFERENCES

1. FEYNMAN, R. P. Development of the space-time view of quantum electrodynamics. *Science 153*, No. 3737 (Aug. 1966), 699-708.
2. SHANNON, C. E. A universal Turing machine with two internal states. In *Automata Studies*, Shannon, C. E., and McCarthy, J. (Eds.), Princeton U. Press, Princeton, N. J., 1956, pp. 157-165.
3. COOK, S. A. On the minimum computation time for multiplication. Doctoral diss., Harvard U., Cambridge, Mass., 1966.
4. KNUTH, D. *The Art of Computer Programming, Vol. II*. Addison-Wesley, Reading, Mass., 1969.
5. MINSKY, M., AND PAPERT, S. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, Mass., 1969.
6. GUZMÁN, A., AND MCINTOSH, H. V. CONVERT. *Comm. ACM 9*, 8 (Aug. 1966), 604-615.
7. HEWITT, C. PLANNER: A language for proving theorems in robots. In: Proc. of the International Joint Conference on Artificial Intelligence, May 7-9, 1969, Washington, D. C., Walker, D. E., and Norton, L. M. (Eds.), pp. 295-301.
8. LEVIN, M., ET AL. *The LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Mass., 1965.
9. WEISSMAN, CLARK. *The LISP 1.5 Primer*. Dickenson Pub. Co., Belmont, Calif., 1967.
10. MARTIN, W. A. Symbolic mathematical laboratory. Doctoral diss., MIT, Cambridge, Mass., Jan. 1967.
11. MOSES, J. Symbolic integration. Doctoral diss., MIT, Cambridge, Mass., Dec. 1967.
12. SLAGLE, J. R. A heuristic program that solves symbolic integration problems in Fresh-

- man calculus. In *Computers and Thought*, Feigenbaum, E. A., and Feldman, J. (Eds.), McGraw-Hill, New York, 1963.
13. TURING, A. M. Computing machinery and intelligence. *Mind* 59 (Oct. 1950), 433-460; reprinted in *Computers and Thought*, Feigenbaum, E. A., and Feldman, J. (Eds.), McGraw-Hill, New York, 1963.
 14. MINSKY, M. Matter, mind and models. Proc. IFIP Congress 65, Vol. 1, pp. 45-49 (Spartan Books, Washington, D. C.). Reprinted in *Semantic Information Processing*, Minsky, M. (Ed.), MIT Press, Cambridge, Mass., 1968, pp. 425-432.
 15. FEIGENBAUM, E. A., AND FELDMAN, J. *Computers and Thought*. McGraw-Hill, New York, 1963.
 16. MINSKY, M. (Ed.). *Semantic Information Processing*. MIT Press, Cambridge, Mass., 1968.
 17. GALTON, F. *Inquiries into Human Faculty and Development*. Macmillan, New York, 1883.
 18. ATTNEAVE, FRED. Triangles as ambiguous figures. *Amer. J. Psychol.* 81, 3 (Sept. 1968), 447-453.
 19. PAPERT, S. Principes analogues à la récurrence. In *Problèmes de la Construction du Nombre*, Presses Universitaires de France, Paris, 1960.
 20. FEYNMAN, R. P. New textbooks for the "new" mathematics. *Engineering and Science* 28, 6 (March 1965), 9-15 (California Inst. of Technology, Pasadena).
 21. LEHRER, T. New math. In *That Was the Year That Was*, Reprise 6179, Warner Bros. Records.

RECEIVED OCTOBER, 1969; REVISED DECEMBER, 1969

1976 ACM Turing Award Lectures

The 1976 ACM Turing Award was presented jointly to Michael A. Rabin and Dana S. Scott at the ACM Annual Conference in Houston, October 20. In introducing the recipients, Bernard A. Galler, Chairman of the Turing Award Committee, read the following citation:

"The Turing Award this year is presented to Michael Rabin and Dana Scott for individual and joint contributions which not only have marked the course of theoretical computer science, but have set a standard of clarity and elegance for the entire field."

Rabin and Scott's 1959 paper, 'Finite Automata and Their Decision Problems' has become a classic paper in formal language theory that still forms one of the best introductions to the area. The paper is simultaneously a survey and a research article; it is technically simple and mathematically impeccable. It has even been recommended to undergraduates!

In subsequent years, Rabin and Scott have made individual contributions which maintain the standards of their early paper. Rabin's applications of automata theory to logic and Scott's development of continuous semantics for programming languages are two examples of work providing depth and dimension: the first applies computer science to mathematics, and the second applies mathematics to computer science.

Rabin and Scott have shown us how well mathematicians can help a scientist understand his own subject. Their work provides one of the best models of creative applied mathematics."

That was the formal citation, but there is a less formal side to this presentation. I want you to understand that the recipients of this award are real people, doing excellent work, but very much like those of us who are here today. Professor Michael Rabin was born in Germany and emigrated as a small child with his parents to Israel in 1935. He got a MSc. degree in Mathematics from the Hebrew University and later his Ph.D. in Mathematics from Princeton University. After obtaining his Ph.D., he was an H.B. Fine Instructor in Mathematics at Princeton University and Member of the Institute for

Advanced Studies at Princeton. Since 1958 he has been a faculty member at the Hebrew University in Jerusalem. From 1972 to 1975 he was also Rector of the Hebrew University. The Rector is elected by the Senate of the University and is Academic Head of the institution.

Professor Dana S. Scott received his Ph.D. degree at Princeton University in 1958. He has since taught at the University of Chicago, the University of California at Berkeley, Stanford University, University of Amsterdam, Princeton University and Oxford University. His present title is Professor of Mathematical Logic at Oxford University in England.

Professor Rabin will speak on "Computational Complexity," and Professor Scott will speak on "Logic and Programming Languages."

Rabin's paper begins below; Scott's paper begins on page 634.

Michael O. Rabin

Dana S. Scott



Complexity of Computations

Michael O. Rabin
Hebrew University of Jerusalem

The framework for research in the theory of complexity of computations is described, emphasizing the interrelation between seemingly diverse problems and methods. Illustrative examples of practical and theoretical significance are given. Directions for new research are discussed.

Key Words and Phrases: complexity of computations, algebraic complexity, intractable problems, probabilistic algorithms

CR Categories: 5.25

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address: Department of Mathematics, Hebrew University of Jerusalem, Jerusalem, Israel.

1. Introduction

The theory of complexity of computations addresses itself to the quantitative aspects of the solutions of computational problems. Usually there are several possible algorithms for solving a problem such as evaluation of an algebraic expression, sorting a file, or parsing a string of symbols. With each of the algorithms there are associated certain significant cost functions such as the number of computational steps as a function of the problem size, memory space requirements for the computation, program size, and in hardware implemented algorithms, circuit size and depth.

The following questions can be raised with respect to a given computational problem P . What are good algorithms for solution of the problem P ? Can one establish and prove a lower bound for one of the cost functions associated with the algorithm? Is the problem perhaps intractable in the sense that no algorithm will solve it in practically feasible time? These questions can be raised for worst-case behavior as well as for the average behavior of the algorithms for P . Another distinction is the one between sequential and parallel algorithms for P . During the last year an extension of algorithms to include randomization within the computation was proposed. Some of the above considerations can be generalized to these probabilistic algorithms.

These questions concerning complexity were the subject of intensive study during the last two decades both within the framework of a general theory and for specific problems of mathematical and practical importance. Of the many achievements let us mention the Fast Fourier Transform, recently significantly improved, with its manifold applications including those to communications:

- Showing that some of the mechanical theorem proving problems arising in proving the correctness of programs, are intractable;
- Determining the precise circuit complexity needed for addition of n -bit numbers;
- Surprisingly fast algorithms for combinatorial and graph problems and their relation to parsing;
- Considerable reductions in computing time for certain important problems, resulting from probabilistic algorithms.

There is no doubt that work on all the above-mentioned problems will continue. In addition we see for the future the branching out of complexity theory into important new areas. One is the problem of secure communication, where a new, strengthened theory of complexity is required to serve as a firm foundation. The other is the investigation of the cost functions pertaining to data structures. The enormous size of the contemplated databases calls for a deeper understanding of the inherent complexity of processes such as the construction and search of lists. Complexity theory provides the point of view and the tools necessary for such a development.

The present article, which is an expanded version of the author's 1976 Turing lecture, is intended to give the reader a bird's-eye view of this vital field. We shall focus our attention on highlights and on questions of methodology, rather than attempt a comprehensive survey.

2. Typical Problems

We start with listing some representative computational problems which are of theoretical and often also of practical importance, and which were the subject of intensive study and analysis. In subsequent sections we shall describe the methods brought to bear on these problems, and some of the important results obtained.

2.1 Computable Functions from Integers to Integers

Let us consider functions of one or more variables from the set $N = \{0, 1, 2, \dots\}$ of integers into N . We recognize intuitively that functions such as $f(x) = x!$, $g(x, y) = x^y + y^x$ are computable.

A.M. Turing, after whom these lectures are so aptly named, set for himself the task of defining in precise terms which functions $f: N \rightarrow N$, $g: N \times N \rightarrow N$, etc. are effectively computable. His model of the idealized computer and the class of recursive functions calcul-

able by this computer are too well known to require exposition.

What concerns us here is the question of measurement of the amount of computational work required for finding a value $f(n)$ of a computable function $f: N \rightarrow N$. Also, is it possible to exhibit functions which are difficult to compute by every program? We shall return to these questions in 4.1.

2.2 Algebraic Expressions and Equations

Let $E(x_1, \dots, x_n)$ be an algebraic expression constructed from the variables x_1, \dots, x_n by the arithmetical operations $+, -, *, /$. For example, $E = (x_1 + x_2) * (x_3 + x_4) / x_1 * x_5$. We are called upon to evaluate $E(x_1, \dots, x_n)$ for a numerical substitution $x_1 = c_1, \dots, x_n = c_n$. More generally, the task may be to evaluate k expressions $E_1(x_1, \dots, x_n), \dots, E_k(x_1, \dots, x_n)$, for the simultaneous substitution $x_1 = c_1, \dots, x_n = c_n$.

Important special cases are the following. Evaluation of a polynomial

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0. \quad (1)$$

Matrix multiplication AB , where A and B are $n \times n$ matrices. Here we have to find the values of the n^2 expressions $a_{ij}b_{1j} + \dots + a_{ij}b_{nj}$, $1 \leq i, j \leq n$, for given numerical values of the a_{ij}, b_{ij} .

Our example for the solution of equations is the system

$$a_1 x_1 + \dots + a_n x_n = b_i, \quad 1 \leq i \leq n, \quad (2)$$

for n linear equations in n unknowns x_1, \dots, x_n . We have to solve (evaluate) the unknowns, given the coefficients a_{ij}, b_{ij} , $1 \leq i, j \leq n$.

We shall not discuss here the interesting question of *approximate solutions* for algebraic and transcendental equations, which is also amenable to the tools of complexity theory.

2.3 Computer Arithmetic

Addition. Given two n -digit numbers $a = \alpha_{n-1}\alpha_{n-2}\dots\alpha_0, b = \beta_{n-1}\beta_{n-2}\dots\beta_0$ (e.g. for $n = 4$, $a = 1011, b = 1100$), to find the $n + 1$ digits of the sum $a + b = \gamma_n\gamma_{n-1}\dots\gamma_0$.

Multiplication. For the above a, b , find the $2n$ digits of the product $a * b = \delta_{2n}\delta_{2n-1}\dots\delta_0$.

The implementation of these arithmetical tasks may be in *hardware*. In this case the base is 2, and $\alpha_i, \beta_i = 0, 1$. Given a fixed n we wish to construct a circuit with $2n$ inputs and, for addition, $n + 1$ outputs. When the $2n$ bits of a, b enter as inputs, the $n + 1$ outputs will be $\gamma_n, \gamma_{n-1}, \dots, \gamma_0$. Similarly for multiplication.

Alternatively we may think about implementation of arithmetic by an algorithm, i.e. in *software*. The need for this may arise in a number of ways. For example, our arithmetical unit may perform just addition, multiplication must then be implemented by a subroutine.

Implementation of arithmetic by a program also

comes up in the context of *multiprecision arithmetic*. Our computer has word size k and we wish to add and multiply numbers of length nk (n -word numbers). We take as base the number 2^k , so that $0 \leq \alpha_i, \beta_i < 2^k$, and use algorithms for finding $a + b$, $a * b$.

2.4 Parsing Expressions in Context-Free Languages

The scope of complexity theory is by no means limited to algebraic or arithmetical calculations. Let us consider *context-free grammars* of which the following is an example. The *alphabet* of G consists of the symbols $t, x, y, z, (,), +, *$. Of these symbols t is a *nonterminal* and all the other symbols are *terminals*. The *productions* (or rewrite rules) of G are

1. $t \rightarrow (t + t)$,
2. $t \rightarrow t * t$,
3. $t \rightarrow x$,
4. $t \rightarrow y$,
5. $t \rightarrow z$.

Starting from t , we can successively rewrite words by use of the productions. For example,

$$t \xrightarrow{1} (t + t) \xrightarrow{3} (x + t) \xrightarrow{2} (x + t * t) \xrightarrow{4} (x + y * t) \xrightarrow{5} (x + y * z). \quad (3)$$

The number above each arrow indicates the production used, and t stands for the nonterminal to be rewritten. A sequence such as (3) is called a *derivation*, and we say that $(x + y * z)$ is derivable from t . The set of all words u derivable from t and containing only terminals, is called the *language generated* by G and is denoted by $L(G)$. The above G is just an example, and the generalization to arbitrary context-free grammars is obvious.

Context-free grammars and languages commonly appear in programming languages and, of course, also in the analysis of natural languages. Two computational problems immediately come up. Given a grammar G and a word W (i.e. string of symbols) on the alphabet of G , is $W \in L(G)$? This is the *membership* problem.

The *parsing problem* is the following. Given a word $W \in L(G)$, find a derivation sequence by productions of G , similar to (3), of W from the initial symbol of G . Alternatively, we want a *parse tree*, of W . Finding a parse tree of an algebraic expression, for example, is an essential step in the compilation process.

2.5 Storing of Files

A file of records R_1, R_2, \dots, R_n is stored in either secondary or main memory. The index i of the record R_i indicates its location in memory. Each record R has a key (e.g. the social-security number in a income-tax file) $k(R)$. The computational task is to rearrange the file in memory into a sequence R_{i_1}, \dots, R_{i_n} so that the keys are in ascending order

$$k(R_{i_1}) < k(R_{i_2}) < k(R_{i_n}).$$

We emphasize both the distinction between the key and the record, which may be considerably larger than the key, and the requirement of actually rearranging the records. These features make the problem more realistic and somewhat harder than the mere sorting of numbers.

2.6 Theorem Proving by Machine

Ever since the advent of computers, trying to endow them with some genuine powers of reasoning was an understandable ambition resulting in considerable efforts being expended in this direction. In particular, attempts were made to enable the computer to carry out logical and mathematical reasoning, and this by proving theorems of pure logic or by deriving theorems of mathematical theories. We consider the important example of the theory of addition of natural numbers.

Consider the system $\mathcal{N} = \langle N, + \rangle$ consisting of the natural numbers $N = \{0, 1, \dots\}$ and the operation $+$ of addition. The formal language L employed for discussing properties of \mathcal{N} is a so-called first-order predicate language. It has variables x, y, z, \dots ranging over natural numbers, the operation symbol $+$, equality $=$, the usual propositional connectives, and the quantifiers \forall ("for all") and \exists ("there exists").

A sentence such as $\exists x \forall y [x + y = y]$ is a formal transcription of "there exists a number x so that for all numbers y , $x + y = y$." This sentence is in fact true in \mathcal{N} .

The set of all sentences of L true in \mathcal{N} will be called the *theory* of \mathcal{N} ($Th(\mathcal{N})$) and will be denoted by $PA = Th(\mathcal{N})$. For example,

$$\forall x \forall y \exists z [x + z = y \vee y + z = x] \in PA.$$

We shall also use the name "Presburger's arithmetic," honoring Presburger, who has proved important results about $Th(\mathcal{N})$.

The *decision problem* for PA is to find an algorithm, if indeed such an algorithm exists, for determining for every given sentence F of the language L whether $F \in PA$ or not.

Presburger [12] has constructed such an algorithm for PA . Since his work, several researchers have attempted to devise efficient algorithms for this problem and to implement them by programs. These efforts were often within the framework of projects in the area of automated programming and program verification. This is because the properties of programs that one tries to establish are sometimes reducible to statements about the addition of natural numbers.

3. Central Issues and Methodology of Computational Complexity

In the previous section we listed some typical computational tasks. Later we shall present results which were obtained with respect to these problems. We shall now describe in general terms the main questions that are raised, and the central concepts that play a role in complexity theory.

3.1 Basic Concepts

A class of similar computational tasks will be called a *problem*. The individual cases of a problem P are

called *instances* of P . Thus P is the set of all its instances. The delineation of a problem is, of course, just a matter of agreement and notational convenience. We may, for example, talk about the problem of matrix multiplication. The instances of this problem are, for any integer n , the pairs A, B of $n \times n$ matrices which are to be multiplied.

With each instance $I \in P$ of a problem P we associate a size, usually an integer, $|I|$. The size function $|I|$ is not unique and its choice is dictated by the theoretical and practical considerations germane to the discussion of the problem at hand.

Returning to the example of matrix multiplication, a reasonable measure on a pair $I = (A, B)$ of $n \times n$ matrices to be multiplied, is $|I| = n$. If we study memory space requirement for an algorithm for matrix multiplication, then the measure $|I| = 2n^2$ may be appropriate. By way of contrast, it does not seem that the size function $|I| = n^n$ would naturally arise in any context.

Let P be a problem and AL an algorithm solving it. The algorithm AL executes a certain computational sequence S_I when solving the instance $I \in P$. With S_I we associate certain measurements. Some of the significant measurements are the following: (1) The length of S_I , which is indicative of computation time. (2) Depth of S_I , i.e., the number of layers of concurrent steps into which S_I can be decomposed. Depth corresponds to the time S_I would require under parallel computation. (3) The memory space required for the computation S_I . (4) Instead of total number of steps in S_I , we may count the number of steps of a certain kind such as arithmetical operations in algebraic computations, number of comparisons in sorting, or number of fetches from memory.

For hardware implementations of algorithms, we usually define the size $|I|$ so that all instances I of the same size n are to be solved on one circuit C_n . The complexity of a circuit C is variously defined as number of gates; depth, which is again related to computing time; or other measurements, such as number of modules, having to do with the technology used for implementing the circuit.

Having settled on a measure μ on computations S , a complexity of computation function F_{AL} can be defined in a number of ways, the principal two being *worst-case* complexity and *average-behavior* complexity. The first notion is defined by

$$F_{AL}(n) = \max \{ \mu(S_I) \mid I \in P, |I| = n \}. \quad (4)$$

In order to define average behavior we must assume a probability distribution p on each set $P_n = \{I \mid I \in P, |I| = n\}$. Thus for $I \in P$, $|I| = n$, $p(I)$ is the probability of I arising among all other instances of size n . The *average behavior* of AL is then defined by

$$M_{AL}(n) = \sum_{I \in P_n} p(I) \mu(S_I). \quad (5)$$

We shall discuss in 4.7 the applicability of the assumption of a probability distribution.

The *analysis* of algorithms deals with the following question. Given a size-function $|I|$ and a measure $\mu(S_I)$ on computations, to exactly determine for a given algorithm AL solving a problem P either the worst-case complexity $F_{AL}(n)$ or, under suitable assumptions, the average behavior $M_{AL}(n)$. In the present article we shall not enter upon questions of analysis, but rather assume that the complexity function is known or at least sufficiently well determined for our purposes.

3.2 The Questions

We have now at our disposal the concepts needed for posing the central question of complexity theory: Given a computational problem P , how well, or at what cost, can it be solved? We do not mention any specific algorithm for solving P . We rather aim at surveying *all* possible algorithms for solving P and try to make a statement concerning the inherent computational complexity of P .

It should be borne in mind that a preliminary step in the study of complexity of a problem P is the choice of the measure $\mu(s)$ to be used. In other words, we must decide on mathematical or practical grounds, *which* complexity we want to investigate. Our study proceeds once this choice was made.

In broad lines, with more detailed examples and illustrations to come later, here are the main issues that will concern us. With the exception of the last item, they seem to fall into pairs.

- (1) Find efficient algorithms for the problem P .
- (2) Establish lower bounds for the inherent complexity of P .
- (3) Search for exact solutions of P .
- (4) Algorithms for approximate (near) solutions.
- (5) Study of worst-case inherent complexity.
- (6) Study of the average complexity of P .
- (7) Sequential algorithms for P .
- (8) Parallel-processing algorithms for P .
- (9) Software algorithms.
- (10) Hardware-implemented algorithms.
- (11) Solution by probabilistic algorithms.

Under (1) we mean the search for good practical algorithms for a given problem. The challenge stems from the fact that the immediately obvious algorithms are often replaceable by much superior ones. Improvements by a factor of 100 are not unheard of. But even a saving of half the cost may sometimes mean the difference between feasibility and nonfeasibility.

While any one algorithm AL for P yields an *upper bound* $F_{AL}(n)$ to the complexity of P , we are also interested in *lower bounds*. The typical result states that every AL solving P satisfies $g(n) \leq F_{AL}(n)$, at least for $n_0 < n$ where $n_0 = n_0(AL)$. In certain happy circumstances upper bounds meet lower bounds. The complexity for such a problem is then completely known. In any case, besides the mathematical interest in lower bounds, once a lower bound is found it guides us in the search

for good algorithms by indicating which efficiencies should not be attempted.

The idea of near-solutions (4) for a problem is significant because sometimes a practically satisfactory near-solution is much easier to calculate than the exact solution.

The main questions (1) and (2) can be studied in combination with one or more of the alternatives (3)-(11). Thus, for example, we can investigate an upper bound for the average time required for sorting by k processors working in parallel. Or we may study the number of logical gates needed for sorting n input-bits.

It would seem that with the manifold possibilities of choosing the complexity measure and the variety of questions that can be raised, the theory of complexity of computations would become a collection of scattered results and unrelated methods. A theme that we try to stress in the examples we present is the large measure of coherence within this field and the commonality of ideas and methods that prevail throughout.

We shall see that efficient algorithms for the parallel evaluation of polynomials, are translatable into circuits for fast addition of n -bit numbers. The Fast Fourier Transform idea yields good algorithms for multiprecision number multiplication. On a higher plane, the relation between software and hardware algorithms mirrors the relation between sequential and parallel computation. Present-day programs are designed to run on a single processor and thus are sequential, whereas a piece of hardware contains many identical subunits which can be viewed as primitive processors operating in parallel. The method of preprocessing appears time and again in our examples, thus being another example of commonality.

4. Results

4.1 Complexity of General Recursive Functions

In [13, 14] the present author initiated the study of classification of computable functions from integers to integers by the complexity of their computation. The framework is axiomatic so that the notions and results apply to every reasonable class of algorithms and every measure on computations.

Let K be a class of algorithms, possibly based on some model of mathematical machines, so that for every computable function $f: N \rightarrow N$ there exists an $AL \in K$ computing it. We do not specify the measure $\mu(S)$ on computations S but rather assume that μ satisfies certain natural axioms. These axioms are satisfied by all the concrete examples of measures listed in 3.1. The size of an integer n is taken to be $|n| = n$. The computation of f is a problem where for each instance n we have to find $f(n)$. Along the lines of 3.1 (4), we have for each algorithm AL for f the complexity of computation function $F_{AL}(n)$ measuring the work involved in computing $f(n)$ by AL .

THEOREM [13, 14]. *For every computable function $g: N \rightarrow N$ there exists a computable function $f: N \rightarrow \{0, 1\}$ so that for every algorithm $AL \in K$ computing f there exists a number n_0 and*

$$g(n) < F_{AL}(n), \quad \text{for } n_0 < n \quad (6)$$

We require that f be a 0-1 valued function because otherwise we could construct a complex function by simply allowing $f(n)$ to grow very rapidly so that writing down the result would be hard.

The limitation $n_0 < n$ in (6) is necessary. For every f and k we can construct an algorithm incorporating a table of the values $f(n)$, $n \leq k$, making the calculation trivial for $n \leq k$.

The main point of the above theorem is that (6), with a suitable $n_0 = n_0(AL)$ holds for every algorithm for f . Thus the inherent complexity of computing f is larger than g .

Starting from [14], E. Blum [1] introduced different but essentially equivalent axioms for the complexity function. Blum obtained many interesting results, including the speed-up theorem. This theorem shows the existence of computable functions for which there is no best algorithm. Rather, for every algorithm for such a function there exists another algorithm computing it much faster.

Research in this area of abstract complexity theory made great strides during the last decade. It served as a basis for the theory of complexity of computations by first bringing up the very question of the cost of a computation, and by emphasizing the need to consider and compare all possible algorithms solving a given problem.

On the other hand, abstract complexity theory does not come to grips with specific computational tasks and their measurement by practically significant yardsticks. This is done in the following examples.

4.2 Algebraic Calculations

Let us start with the example of evaluation of polynomials. We take as our measure the number of arithmetical operations and use the notation (nA, kM) to denote a cost of n additions/subtractions and k multiplications/divisions. By rewriting the polynomial (1) as

$$f(x) = (\dots ((a_n x + a_{n-1})x + a_{n-2}))x + \dots + a_0,$$

we see that the general n -degree polynomial can be evaluated by (nA, nM) . In the spirit of questions 1 and 2 in 3.2, we ask whether a clever algorithm might use fewer operations. Rather delicate mathematical arguments show that the above number is optimal so that this question is completely settled.

T. Motzkin introduced in [9] the important idea of *preprocessing* for a computation. In many important applications we are called upon to evaluate the same polynomial $f(x)$ for many argument values $x = c_1, x = c_2, \dots$. He suggested the following strategy of preprocessing the coefficients of the polynomial (1). Calculate

once and for all numbers $\alpha(a_0, \dots, a_n) \dots \alpha_k(a_0, \dots, a_n)$ from the given coefficients a_0, \dots, a_n . When evaluating $f(c)$ use $\alpha_0, \dots, \alpha_n$. This approach makes computational sense when the cost of preprocessing is small as compared to the total savings in computing $f(c_1), f(c_2), \dots$, i.e. when the expected number of arguments for which $f(x)$ is to be evaluated is large. Motskin obtained the following.

THEOREM. *Using preprocessing, a polynomial of degree n can be evaluated by $(nA, ([n/2] + 2)M)$.*

Again one can prove that this result is essentially the best possible. What about evaluation in parallel? If we use k processors and have to evaluate an expression requiring at least m operations, then the best that we can hope for is computation time $(m/k) \sim 1 + \log_2 k$.

Namely, assume that all processors are continuously busy, then $m - k$ operations are performed in time $(m/k) - 1$. The remaining k operations must combine by binary operations k inputs into one output, and this requires time $\log_2 k$ at least. In view of the above, the following result due to Munro and Paterson [10] is nearly best possible.

THEOREM. *The polynomial (1) can be evaluated by k processors working in parallel in time $(2n/k) + \log_2 k + O(1)$.*

With the advances in hardware it is not unreasonable to expect that we may be able to employ large numbers of processors on the same task. Brent [3], among others, studied the implications of unlimited parallelism and proved the following.

THEOREM. *Let $E(x_1, \dots, x_n)$ be an arithmetical expression, where each variable x_i appears only once. The expression E can be evaluated under unlimited parallelism in time $4 \log_2 n$.*

Another important topic is the Fast Fourier Transform (FFT). The operation of convolution which has many applications such as to signal processing, is an example of a computation greatly facilitated by the FFT. Let a_1, \dots, a_n be a sequence of n numbers, b_1, b_2, \dots , be a stream of incoming numbers. Define for $i = 1, 2, \dots$,

$$c_i = a_1 b_i + a_2 b_{i+1} + \dots + a_n b_{i+n-1}. \quad (7)$$

We have to calculate the values c_1, c_2, \dots . From (7) it seems that the cost per value of c_i is $2n$ operations. If we compute the c_i 's in blocks of size n , i.e. c_1, \dots, c_n , and c_{n+1}, \dots, c_{2n} , etc. using FFT, then the cost per block is about $4n \log_2 n$ so that the cost of a single c_i is $4 \log_2 n$.

Using a clever combination of algebraic and number-theoretic ideas, S. Winograd [20] recently improved computation times of convolution for small values of n and of the discrete Fourier transform for small to medium values of n . For $n \sim 1000$, for example, his method is about twice as fast as the conventional FFT algorithm.

The obvious methods for $n \times n$ matrix multiplication and for the solution of the system (2) of n linear

equations in n unknowns require about n^3 operations. Strassen [17] found the following surprising result.

THEOREM. *Two $n \times n$ matrices can be multiplied using at most $4.7n^{2.81}$ operations. A system of n linear equations in n unknowns can be solved by $4.8n^{2.81}$ operations.*

It is not likely that the exponent $\log_2 7 \sim 2.81$ is really the best possible, but at the time of writing of this article all attempts to improve this result have failed.

4.3 How Fast Can We Add or Multiply?

This obviously important question underwent thorough analysis. A simple fan-in argument shows that if gates with r inputs are used, then a circuit for the addition of n -bit numbers requires at least time $\log_r n$. This lower bound is in fact achievable.

It is worthwhile noticing that, in the spirit of the remarks in 3.2 concerning the analogy between parallel algorithms and hardware algorithms, one of the best results on circuits for addition (Brent [2]) employs Boolean identities which are immediately translatable into an efficient parallel evaluation algorithm for polynomials.

The above results pertain to the binary representation of the numbers to be added. Could it be that under a suitably clever coding of the numbers $0 \leq a < 2^n$, addition mod 2^n is performable in time less than $\log_r n$? Winograd [19] answered this question. Under very general assumptions on the coding, the lower bound remains $\log_r n$.

Turning to multiprecision arithmetic, the interesting questions arise in connection with multiplication. The obvious method for multiplying numbers of length n involves n^2 bit-operations. Early attempts at improvements employed simple algebraic identities and resulted with a reduction to $O(n^{1.58})$ operations.

Schönhage and Strassen [16] utilized the connection between multiplication of natural numbers and polynomial multiplication and employed the FFT to obtain the following theorem.

THEOREM. *Two n -bit numbers can be multiplied by $O(n \log n \log \log n)$ bit-operations.*

Attempts at lower bounds for complexity of integer multiplication must refer to a specific computation model. Under very reasonable assumptions Paterson Fischer and Meyer [11] have considerably narrowed the gap between the upper and lower bounds by showing the following.

THEOREM. *At least $O(n \log n / \log \log n)$ operations are necessary for multiplying n -bit numbers.*

4.4 Speed of Parsing

Parsing of expressions in context-free grammars would seem at first sight to require a costly backtracking computation. A dynamical computation which simultaneously seeks the predecessors of all substrings of the string to be parsed, leads to an algorithms requiring $O(n^3)$ steps for parsing a word of length n . The coeffi-

cient of n^3 depends on the grammar. This was for a long while the best result, even though for special classes of context-free grammars better upper bounds were obtained.

Fischer and Meyer observed that Strassen's algorithm for matrix multiplication can be adapted to yield an $O(n^{2.81}c(n))$ bit-operations algorithm for the multiplication of two $n \times n$ boolean matrices. Here $c(n) = \log n \log \log n \log \log \log n$ and is thus $O(n^\alpha)$ for every $0 < \alpha$.

Valiant [18] found that parsing is at most as complex as boolean matrix multiplication. Hence, since actually $\log_2 7 < 2.81$, the following theorem holds:

THEOREM. *Expressions of length n in the context-free language $L(G)$ can be parsed in time $d(G)n^{2.81}$.*

We again see how results from algebraic complexity bear fruit in the domain of complexity of combinatorial computations.

4.5 Data Processing

Of the applications of complexity theory to data processing we discuss the best known example, that of sorting. We follow the formulation given in 2.5.

It is well known that the sorting of n numbers in random access memory requires about $n \log n$ comparisons. This is both the worst-case behavior of some algorithms and the average behavior of other algorithms under the assumption that all permutations are equally likely.

The rearrangement of records R_1, R_2, \dots, R_n , poses additional problems because the file usually resides in some sequential or nearly sequential memory such as magnetic tape or disk. Size limitations enable us to transfer into the fast memory for rearrangement only a small number of records at a time. Still it is possible to develop algorithms for the actual reordering of the files in time $cn \log n$ where c depends on characteristics of the system under discussion.

An instructive result in this area is due to Floyd [6]. In his model the file is distributed on a number of pages P_1, \dots, P_m and each page contains k records so that P_i contains the records R_{i1}, \dots, R_{ik} . For our purposes we may assume without loss of generality that $m = k$. The task is to redistribute the records so that R_{ij} will go to page P_j for all $1 \leq i, j \leq k$. The fast memory is large enough to allow reading in two pages P_e, P_f , redistribute their records and read the pages out. Using a recursion analogous to the one employed in the FFT, Floyd proved the following.

THEOREM. *The redistribution of records in the above manner can be achieved by $k \log_2 k$ transfers into fast memory. This result is the best possible.*

The lower bound is established by considering a suitable entropy function. It applies under the assumption that within fast memory the records are just shuffled. It is not known whether allowing computations with the records, viewed as strings of bits, may produce an algorithm with fewer fetches of pages.

4.6 Intractable Problems

The domain of theorem proving by machine serves as a source of computational problems which require such an inordinate number of computational steps so as to be intractable. In attempts to run programs for the decision problem of Presburger's arithmetic (PA) on the computer, the computation terminated only on the simplest instances tried. A theoretical basis for this pragmatic fact is provided by the following result due to Fischer and Rabin [5].

THEOREM. *There exists a constant $\theta < c$ so that for every decision algorithm AL for PA there is a number n_0 such that every $n_0 < n$ there is a sentence H of the language L (the language for addition of numbers) satisfying (1) $I(H) = n$, (2) AL takes more than 2^{2^n} steps to determine whether $H \in PA$, i.e. whether H is true in $\langle N, + \rangle$.*

The constant c depends on the notation used for stating properties of $\langle N, + \rangle$. In any case, it is not very small. The rapid growth of the inherent lower bound 2^{2^n} shows that even when trying to solve the decision problem for this very simple and basic mathematical theory, we run into practically impossible computations. Meyer [8] produced examples of theories with even more devastatingly complex decision problems.

The simplest level of logical deduction is the propositional calculus. From propositional variables p_1, p_2, \dots , we can construct formulas such as $[p_1 \wedge \sim p_1] \vee [p_2 \wedge \sim p_1]$ by the propositional connectives. The *satisfiability problem* is to decide for a propositional formula $G(p_1, \dots, p_n)$ whether there exists a truth-value assignment to the variables p_1, \dots, p_n so that G becomes true. The assignment $p_1 = F$ (false), $p_2 = T$ (true), for example, satisfies the above formula.

The straightforward algorithm for the satisfiability problem will require about 2^n steps for a formula with n variables. It is not known whether there exist nonexponential algorithms for the satisfiability problem.

The great importance of this question was brought to the forefront by Cook [4]. One can define a natural process of so called polynomial reduction of one computational problem P to another problem Q . If P is polynomially reducible to Q and Q is solvable in polynomial time then so is P . Two problems which are mutually reducible are called polynomially equivalent. Cook has shown that the satisfiability problem is equivalent to the so called problem of cliques in graphs. Karp [7] brings a large number of problems equivalent to satisfiability. Among them the problems of 0-1 integer programming, the existence of Hamiltonian circuits in a graph, and the integer-valued traveling-salesman problem, to mention just a few examples.

In view of these equivalences, if any one of these important problems is solvable in polynomial time then so are all the others. The question whether satisfiability is of polynomial complexity is called the $P = NP$ problem and is justly the most celebrated problem in the theory of complexity of computations.

4.7 Probabilistic Algorithms

As mentioned in 3.1, the study of the average behavior, or expected time, of an algorithm is predicated on the assumption of a probability distribution on the space of instances of the problem. This assumption involves certain methodological difficulties. We may postulate a certain distribution such as all instances being equally likely, and in a practical situation the source of instances of the problem to be solved may be biased in an entirely different way. The distribution may be shifting with time and will often not be known to us. In the extreme case, most instances which actually come up are precisely those for which the algorithm behaves worst.

Could we employ probability in computations in a different manner, one over which we have total control? A *probabilistic algorithm* AL for a problem P uses a source of random numbers. When solving an instance $I \in P$, a short sequence $r = (b_1, \dots, b_k)$ of random numbers is generated, and these are used in AL to solve P in *exact terms*. With the exception of the random choice of r , the algorithm proceeds completely deterministically.

We say that such an AL solves P in expected time $f(n)$ if for every $I \in P$, $|I| = n$, AL solves I in expected time less or equal to $f(n)$. By expected time we mean the average of all solution times of I by AL for all possible choice sequences r (which we assume to be equally likely).

Let us notice the difference between this notion and the well known Monte-Carlo method. In the latter method we construct for a problem a stochastic process which emulates it and then measure the stochastic process to obtain an approximate solution for the problem. Thus the Monte-Carlo method is, in essence, an analog method of solution. Our probabilistic algorithms, by contrast, use the random numbers b_1, \dots, b_k to determine branchings in otherwise deterministic algorithms and produce exact rather than approximate solutions.

It may seem unlikely that such a consultation with a "throw of the dice" could speed up a computation. The present author systematically studied [15] probabilistic algorithms. It turns out that in certain cases this approach effects dramatic improvements.

The nearest pair in a set of points $x_1, \dots, x_n \in R^k$ (k -dimensional space) is the pair x_i, x_j , $i \neq j$, for which the distance $d(x_i, x_j)$ is minimal. A probabilistic algorithm finds the nearest pair in expected time $O(n)$, more rapidly than any conventional algorithm.

The problem of determining whether a natural number n is prime becomes intractable for large n . The present methods break down around $n \sim 10^{60}$ when applied to numbers which are not of a special form. A probabilistic algorithm devised by the author works in time $O(\log n)^3$. On a medium-sized computer, $2^{400} - 593$ was recognized as prime within a few minutes. The method works just as well on any other number of comparable size.

The full potential of these ideas is not yet known and merits further study.

5. New Directions

Of the possible avenues for further research, let us mention just two.

5.1 Large Data Structures

Commercial needs prescribe the creation of ever larger databases. At the same time present-day and, even more so imminent future technologies, make it possible to create gigantic storage facilities with varying degrees of freedom of access.

Much of the current research on databases is directed at the interface languages between the user and the system. But the enormous sizes of the lists and other structures contemplated would tend to make the required operations on these structures very costly unless a deeper understanding of the algorithms to perform such operations is gained.

We can start with the problem of finding a theoretical, but at the same time practically significant, model for lists. This model should be versatile enough to enable specialization to the various types of list structures now in use.

What are operations on lists? We can enumerate a few. Search through a list, garbage collection, access to various points in a list, insertions, deletions, mergers of lists. Could one systematize the study of these and other significant operations? What are reasonable cost functions that can be associated with these operations?

Finally, a deep quantitative understanding of data structures could be a basis for recommendations as to technological directions to be followed. Does parallel processing appreciably speed up various operations on data structures? What useful properties can lists be endowed with in associative memories? These are, of course, just examples.

5.2 Secure Communications

Secure communications employ some kind of coding devices, and we can raise fundamental questions of complexity of computations in relation to these systems. Let us illustrate this by means of the system of block-encoding.

In block-encoding one employs a digital device which takes as inputs words of length n and encodes them by use of a key. If x is a word of length n and z is a key (let us assume that keys are also of length n), then let $E_z(x) = y$, $I(y) = n$, denote the result of encoding x by use of the key z . A message $w = x_1, x_2, \dots, x_k$ of length kn is encoded as $E_z(x_1) E_z(x_2) \dots E_z(x_k)$.

If an adversary is able to obtain the current key z , then he can decode the communications between the parties since we assume that he is in possession of the coding and decoding equipment. He can also interject

into the line bogus messages which will decode properly. In commercial communications this possibility is perhaps even more dangerous than the breach of security.

In considering security one should take into account the possibility that the adversary gets hold of a number of messages w_1, w_2, \dots , in clear text, and in encoded form $E_z(w_1), E_z(w_2), \dots$. Can the key z be computed from this data?

It would not do to prove that such a computation is intractable. For the results of current complexity theory give us worst-case information. Thus if, say, for the majority of key-retrieval computations a lower bound of 2^n on computational complexity will be established, then the problem will be deemed intractable. But if an algorithm will discover the key in practical time in one case in a thousand then the possibilities of fraud would be unacceptably large.

Thus we need a theory of complexity that will enable us to state and prove that a certain computation is intractable in virtually every case. For example, a block-encoding system is safe if any algorithm for key-determination will terminate in practical time only on $O(2^{cn})$ of the cases. We are very far from creation of such a theory, especially at the present stage when $P = NP$ is not yet settled.

References

1. Blum, M. A machine independent theory of the complexity of recursive functions. *J. ACM* 14 (1967), 322-336.
2. Brent, R.P. On the addition of binary numbers. *IEEE Trans. Comptrs. C-19* (1970), 758-759.
3. Brent, R.P. The parallel evaluation of algebraic expressions in logarithmic time. *Complexity of Sequential and Parallel Numerical Algorithms*, J.F. Traub, Ed., Academic Press, New York, 1973, pp. 83-102.
4. Cook, S.A. The complexity of theorem proving procedures. *Proc. Third Annual ACM Symp. on Theory of Computng.*, 1971, pp. 151-158.
5. Fischer, M.J., and Rabin, M.O. Super-exponential complexity of Presburger arithmetic. In *Complexity of Computations* (SIAM-AMS Proc., Vol. 7), R.M. Karp Ed., 1974, pp. 27-41.
6. Floyd, R.W. Permuting information in idealized two-level storage. In *Complexity of Computer Computations*, R. Miller and J. Thatcher Eds., Plenum Press, New York, 1972, pp. 105-109.
7. Karp, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher Eds., Plenum Press, New York, 1972, pp. 85-103.
8. Meyer, A.R. The inherent computational complexity of theories of order. *Proc. Int. Cong. Math.*, Vol. 2, Vancouver, 1974, pp. 477-482.
9. Motzkin, T.S. Evaluation of polynomials and evaluation of rational functions. *Bull. Amer. Math. Soc.* 61 (1955), 163.
10. Munro, I., and Paterson, M. Optimal algorithms for parallel polynomial evaluation. *J. Compr. Syst. Sci.*, 7 (1973), 189-198.
11. Paterson, M., Fischer, M.J., and Meyer, A.R. An improved overlap argument for on-line multiplication. Proj. MAC Tech. Report 40, M.I.T. (1974).
12. Presburger, M. Über die Vollständigkeit eines gewissen Systems Arithmetic ganzer Zahlen in welchem die Addition als einzige Operation hervortritt. *Comptes-rendus du 1 Congrès de Mathématiciens des Pays Slaves*, Warsaw, 1930, pp. 92-101, 395.
13. Rabin, M.O. Speed of computation and classification of recursive sets. *Third Convention Sci. Soc., Israel*, 1959, pp. 1-2.
14. Rabin, M.O. Degree of difficulty of computing a function and a partial ordering of recursive sets. *Tech. Rep. No. 1, O.N.R.*, Jerusalem, 1960.
15. Rabin, M.O. Probabilistic algorithms. In *Algorithms and Complexity, New Directions and Recent Trends*, J.F. Traub Ed., Academic Press, New York, 1976, pp. 21-39.
16. Schönhage, A., and Strassen, V. Schnelle Multiplication grosser Zahlen. *Computing* 7 (1971), 281-292.
17. Strassen, V. Gaussian elimination is not optimal. *Num. Math.* 13 (1969), 354-356.
18. Valiant, L.G. General context-free recognition in less than cubic time. *Rep., Dept. Comptr. Sci., Carnegie-Mellon U.*, Pittsburgh, Pa., 1974.
19. Winograd, S. On the time required to perform addition. *J. ACM* 12 (1965), 277-285.
20. Winograd, S. On computing the discrete Fourier transform. *Proc. Natl. Acad. Sci. USA* 73 (1976), 1005-1006.

One Man's View of Computer Science

R. W. HAMMING

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

ABSTRACT. A number of observations and comments are directed toward suggesting that more than the usual engineering flavor be given to computer science. The engineering aspect is important because most present difficulties in this field do not involve the theoretical question of whether certain things can be done, but rather the practical question of how can they be accomplished well and simply.

The teaching of computer science could be made more effective by various alterations, for example, the inclusion of a laboratory course in programming, the requirement for a strong minor in something other than mathematics, and more practical coding and less abstract theory, as well as more seriousness and less game playing.

KEY WORDS AND PHRASES: computer science, computer engineering, practical programming, mathematical game-playing, computer technician, computer professional, true-to-life programming, computer science curriculum, software, basic research, undirected research, programmers' ethical standards, programmers' social responsibility

CR CATEGORIES: 1.3, 1.50

Let me begin with a few personal words. When one is notified that he has been elected the ACM Turing lecturer for the year, he is at first surprised—especially is the nonacademic person surprised by an ACM award. After a little while the surprise is replaced by a feeling of pleasure. Still later comes a feeling of "Why me?" With all that has been done and is being done in computing, why single out me and my work? Well, I suppose that it has to happen to someone each year, and this time I am the lucky person. Anyway, let me thank you for the honor you have given to me and by inference to the Bell Telephone Laboratories where I work and which has made possible so much of what I have done.

The topic of my Turing lecture, "One Man's View of Computer Science," was picked because "What is computer science?" is argued endlessly among people in the field. Furthermore, as the excellent Curriculum 68 report¹ remarks in its introduction, "The Committee believes strongly that a continuing dialogue on the process and goals of education in computer science will be vital in the years to come." Lastly, it is wrong to think of Turing, for whom these lectures were named, as being exclusively interested in Turing machines; the fact is that he contributed to many aspects of the field and would probably have been very interested in the topic, though perhaps not in what I say.

The question "What is computer science?" actually occurs in many different

¹ A Report of the ACM Curriculum Committee on Computer Science; *Comm. ACM* 11, 3 (Mar. 1968), 151-197.

forms, among which are: What is computer science currently? What can it develop into? What should it develop into? What will it develop into?

A precise answer cannot be given to any of these. Many years ago an eminent mathematician wrote a book *What is Mathematics* and nowhere did he try to define mathematics, rather he simply wrote mathematics. While you will now and then find some aspect of mathematics defined rather sharply, the only generally agreed upon definition of mathematics is "Mathematics is what mathematicians do", which is followed by "Mathematicians are people who do mathematics." What is true about defining mathematics is also true about many other fields: there is often no clear, sharp definition of the field.

In the face of this difficulty many people, including myself at times, feel that we should ignore the discussion and get on with *doing* it. But as George Forsythe points out so well² in a recent article, it *does* matter what people in Washington, D. C. think computer science is. According to him, they tend to feel that it is a part of applied mathematics and therefore turn to the mathematicians for advice in the granting of funds. And it is not greatly different elsewhere; in both industry and the universities you can often still see traces of where computing first started, whether in electrical engineering, physics, mathematics, or even business. Evidently the picture which people have of a subject can significantly affect its subsequent development. Therefore, although we cannot hope to settle the question definitively, we need frequently to examine and to air our views on what our subject is and should become.

In many respects, for me it would be more satisfactory to give a talk on some small, technical point in computer science—it would certainly be easier. But that is exactly one of the things that I wish to stress—the danger of getting lost in the details of the field, especially in the coming days when there will be a veritable blizzard of papers appearing each month in the journals. We must give a good deal of attention to a broad training in the field—this in the face of the increasing necessity to specialize more and more highly in order to get a thesis problem, publish many papers, etc. We need to prepare our students for the year 2000 when many of them will be at the peak of their career. It seems to me to be more true in computer science than in many other fields that "specialization leads to triviality."

I am sure you have all heard that our scientific knowledge has been doubling every 15 to 17 years. I strongly suspect that the rate is now much higher in computer science; certainly it was higher during the past 15 years. In all of our plans we must take this growth of information into account and recognize that in a very real sense we face a "semi-infinite" amount of knowledge. In many respects the classical concept of a scholar who knows at least 90 percent of the relevant knowledge in his field is a dying concept. Narrower and narrower specialization is *not* the answer, since in part the difficulty is in the rapid growth of the interrelationships between fields. It is my private opinion that we need to put relatively more stress on quality and less on quantity and that the careful, critical, considered survey articles will often be more significant in advancing the field than new, non-essential material.

We live in a world of shades of grey, but in order to argue, indeed even to think, it is often necessary to dichotomize and say "black" or "white". Of course in doing

² FORSYTHE, G. E. What to do till the computer scientist comes. *Am. Math. Monthly* 75, 5 (May 1968), 454–461.

so we do violence to the truth, but there seems to be no other way to proceed. I trust, therefore, that you will take many of my small distinctions in this light—in a sense, I do not believe them myself, but there seems to be no other simple way of discussing the matter.

For example, let me make an arbitrary distinction between science and engineering by saying that science is concerned with *what* is possible while engineering is concerned with *choosing*, from among the many possible ways, *one* that meets a number of often poorly stated economic and practical objectives. We call the field "computer science" but I believe that it would be more accurately labeled "computer engineering" were not this too likely to be misunderstood. So much of what we do is not a question of can it be done as it is a question of finding a practical way. It is not usually a question of can there exist a monitor system, algorithm, scheduler, or compiler, rather it is a question of finding a practical working one with a reasonable expenditure of time and effort. While I would not change the name from "computer science" to "computer engineering," I would like to see far more of a practical, engineering flavor in what we teach than I usually find in course outlines.

There is a second reason for asking that we stress the practical side. As far into the future as I can see, computer science departments are going to need large sums of money. Now society usually, though not always, is more willing to give money when it can see practical returns than it is to invest in what it regards as impractical activities, amusing games, etc. If we are to get the vast sums of money I believe we will need, then we had better give a practical flavor to our field. As many of you are well aware, we have already acquired a bad reputation in many areas. There have been exceptions, of course, but all of you know how poorly we have so far met the needs for software.

At the heart of computer science lies a technological device, the computing machine. Without the machine almost all of what we do would become idle speculation, hardly different from that of the notorious Scholastics of the Middle Ages. The founders of the ACM clearly recognized that most of what we did, or were going to do, rested on this technological device, and they deliberately included the word "machinery" in the title. There are those who would like to eliminate the word, in a sense to symbolically free the field from reality, but so far these efforts have failed. I do not regret the initial choice. I still believe that it is important for us to recognize that the computer, the information processing machine, is the foundation of our field.

How shall we produce this flavor of practicality that I am asking for, as well as the reputation for delivering what society needs at the time it is needed? Perhaps most important is the way we go about our business and our teaching, though the research we do will also be very important. We need to avoid the bragging of uselessness and the game-playing that the pure mathematicians so often engage in. Whether or not the pure mathematician is right in claiming that what is utterly useless today will be useful tomorrow (and I doubt very much that he is, in the current situation), it is simply poor propaganda for raising the large amounts of money we need to support the continuing growth of the field. We need to avoid making computer science look like pure mathematics: our primary standard for acceptance should be experience in the real world, not aesthetics.

Were I setting up a computer science program, I would give relatively more

emphasis to laboratory work than does Curriculum 68, and in particular I would require every computer science major, undergraduate or graduate, to take a laboratory course in which he designs, builds, debugs, and documents a reasonably sized program, perhaps a simulator or a simplified compiler for a particular machine. The results would be judged on style of programming, practical efficiency, freedom from bugs, and documentation. If any of these were too poor, I would not let the candidate pass. In judging his work we need to distinguish clearly between superficial cleverness and genuine understanding. Cleverness was essential in the past; it is no longer sufficient.

I would also require a strong minor in some field *other* than computer science and mathematics. Without real experience in using the computer to get useful results the computer science major is apt to know all about the marvelous tool except how to use it. Such a person is a mere technician, skilled in manipulating the tool but with little sense of how and when to use it for its basic purposes. I believe we should avoid turning out more idiot savants—we have more than enough “computaiks” now to last us a long time. What we need are professionals!

The Curriculum 68 recognized this need for “true-to-life” programming by saying, “This might be arranged through summer employment, a cooperative work-study program, part-time employment in computer centers, special projects courses, or some other appropriate means.” I am suggesting that the appropriate means is a stiff laboratory course under your own control, and that the above suggestions of the Committee are rarely going to be effective or satisfactory.

Perhaps the most vexing question in planning a computer science curriculum is determining the mathematics courses to require of those who major in the field. Many of us came to computing with a strong background in mathematics and tend automatically to feel that a lot of mathematics should be required of everyone. All too often the teacher tries to make the student into a copy of himself. But it is easy to observe that in the past many highly rated software people were ignorant of most of formal mathematics, though many of them seemed to have a natural talent for mathematics (as it is, rather than as it is so often taught).

In the past I have argued that to require a strong mathematical content for computer science would exclude many of the best people in the field. However, with the coming importance of scheduling and the allocating of the resources of the computer, I have had to reconsider my opinion. While there is some evidence that part of this will be incorporated into the hardware, I find it difficult to believe that there will not be for a long time (meaning at least five years) a lot of scheduling and allocating of resources in software. If this is to be the pattern, then we need to consider training in this field. If we do not give such training, then the computer science major will find that he is a technician who is merely programming what others tell him to do. Furthermore, the kinds of programming that were regarded in the past as being great often depended on cleverness and trickery and required little or no formal mathematics. This phase seems to be passing, and I am forced to believe that in the future a good mathematical background will be needed if our graduates are to do significant work.

History shows that relatively few people can learn much new mathematics in their thirties, let alone later in life; so that if mathematics is going to play a significant role in the future, we need to give the students mathematical training while they are in school. We can, of course, evade the issue for the moment by providing

two parallel paths, one with and one without mathematics, with the warning that the nonmathematical path leads to a dead end so far as further university training is concerned (assuming we believe that mathematics is essential for advanced training in computer science).

Once we grant the need for a lot of mathematics, then we face the even more difficult task of saying specifically which courses. In spite of the numerical analysts' claims for the fundamental importance of their field, a surprising amount of computer science activity requires comparatively little of it. But I believe we can defend the requirement that *every* computer science major take at least one course in the field. Our difficulty lies, perhaps, in the fact that the present arrangement of formal mathematics courses is not suited to our needs as we presently see them. We seem to need some abstract algebra; some queuing theory; a lot of statistics, including the design of experiments; a moderate amount of probability, with perhaps some elements of Markov chains; parts of information and coding theory; and a little on bandwidth and signalling rates, some graph theory, etc., but we also know that the field is rapidly changing and that tomorrow we may need complex variables, topology, and other topics.

As I said, the planning of the mathematics courses is probably the most vexing part of the curriculum. After a lot of thinking on the matter, I currently feel that if our graduates are to make significant contributions and not be reduced to the level of technicians running a tool as they are told by others, then it is better to give them too much mathematics rather than too little. I realize all too well that this will exclude many people who in the past have made contributions, and I am not happy about my conclusion, but there it is. In the future, success in the field of computer science is apt to require a command of mathematics.

One of the complaints regularly made of computer science curriculums is that they seem to almost totally ignore business applications and COBOL. I think that it is not a question of how important the applications are, nor how widely a language like COBOL is used, that should determine whether or not it is taught in the computer science department; rather, I think it depends on whether or not the business administration department can do a far better job than we can, and whether or not what is peculiar to the business applications is fundamental to other aspects of computer science. And what I have indicated about business applications applies, I believe, to most other fields of application that can be taught in other departments. I strongly believe that with the limited resources we have, and will have for a long time to come, we should not attempt to teach applications of computers in the computer science department—rather, those applications should be taught in their natural environments by the appropriate departments.

The problem of the role of analog computation in the computer science curriculum is not quite the same as that of applications to special fields, since there is really no place else for it to go. There is little doubt that analog computers are economically important and will continue to be so for some time. But there is also little doubt that the field, even including hybrid computers, does not have at present the intellectual ferment that digital computation does. Furthermore, the essence of good analog computation lies in the understanding of the physical limitations of the equipment and in the peculiar art of scaling, especially in the time variable, which is quite foreign to the rest of computer science. It tends, therefore, to be ignored rather than to be rejected; it is either not taught or else it is an elective, and

this is probably the best we can expect at present when the center of interest is the general purpose digital computer.

At present there is a flavor of "game-playing" about many courses in computer science. I hear repeatedly from friends who want to hire good software people that they have found the specialist in computer science is someone they do *not* want. Their experience is that graduates of our programs seem to be mainly interested in playing games, making fancy programs that really do not work, writing trick programs, etc. and are unable to discipline their own efforts so that what they say they will do gets done on time and in practical form. If I had heard this complaint merely once from a friend who fancied that he was a hard-boiled engineer, then I would dismiss it; unfortunately I have heard it from a number of capable, intelligent, understanding people. As I earlier said, since we have such a need for financial support for the current and future expansion of our facilities, we had better consider how we can avoid such remarks being made about our graduates in the coming years. Are we going to continue to turn out a product that is not wanted in many places? Or are we going to turn out responsible, effective people who meet the real needs of our society? I hope that the latter will be increasingly true; hence my emphasis on the practical aspects of computer science.

One of the reasons that the computer scientists we turn out are more interested in "cute" programming than in results is that many of our courses are being taught by people who have the instincts of a pure mathematician. Let me make another arbitrary distinction which is only partially true. The pure mathematician starts with the given problem, or else some variant that he has made up from the given problem, and produces what he says is an answer. In applied mathematics it is necessary to add two crucial steps (1) an examination of the relevance of the mathematical model to the actual situation, and (2) the relevance of, or if you wish the interpretation of, the results of the mathematical model back to the original situation. This is where there is the sharp difference: The applied mathematician must be willing to stake part of his reputation on the remark "If you do so and so you will observe such and such very closely and therefore you are justified in going ahead and spending the money, or effort, to do the job as indicated," while the pure mathematician usually shrugs his shoulders and says, "That is none of my responsibility." Someone must take the responsibility for the decision to go ahead on one path or another, and it seems to me that he who does assume this responsibility will get the greater credit, on the average, as it is doled out by society. We need, therefore, in our teaching of computer science, to stress the assuming of responsibility for the *whole* problem and not just the cute mathematical part. This is another reason why I have emphasized the engineering aspects of the various subjects and tried to minimize the purely mathematical aspects.

The difficulty is, of course, that so many of our teachers in computer science are pure mathematicians and that pure mathematics is so much easier to teach than is applied work. There are relatively few teachers available to teach in the style I am asking for. This means we must do the best we can with what we have, but we should be conscious of the direction we want to take and that we want, where possible, to give a practical flavor of responsibility and engineering rather than mere existence of results.

It is unfortunate that in the early stages of computer science it is the talent and ability to handle a sea of minutiae which is important for success. But if the student

is to grow into someone who can handle the larger aspects of computer science, then he must have, and develop, other talents which are not being used or exercised at the early stages. Many of our graduates never make this second step. The situation is much like that in mathematics: in the early years it is the command of the trivia of arithmetic and formal symbol manipulation of algebra which is needed, but in advanced mathematics a far different talent is needed for success. As I said, many of the people in computer science who made their mark in the area where the minutiae are the dominating feature do not develop the larger talents, and they are still around teaching and propagating their brand of detail. What is needed in the higher levels of computer science is not the "black or white" mentality that characterizes so much of mathematics, but rather the judgment and balancing of conflicting aims that characterize engineering.

I have so far skirted the field of software, or, as a friend of mine once said, "ad hoc-ery." There is so much truth in his characterization of software as ad hoc-ery that it is embarrassing to discuss the topic of what to teach in software courses. So much of what we have done has been in an ad hoc fashion, and we have been under so much pressure to get something going as soon as possible that we have precious little which will stand examination by the skeptical eye of a scientist or engineer who asks, "What content is there in software?" How few are the difficult ideas to grasp in the field! How much is mere piling on of detail after detail without any careful analysis! And when 50,000-word compilers are later remade with perhaps 5000 words, how far from reasonable must have been the early ones!

I am no longer a software expert, so it is hard for me to make serious suggestions about what to do in the software field, yet I feel that all too often we have been satisfied with such a low level of quality that we have done ourselves harm in the process. We seem not to be able to use the machine, which we all believe is a very powerful tool for manipulating and transforming information, to do our own tasks in this very field. We have compilers, assemblers, monitors, etc. for others, and yet when I examine what the typical software person does, I am often appalled at how little he uses the machine in his own work. I have had enough minor successes in arguments with software people to believe that I am basically right in my insistence that we should learn to use the machine at almost every stage of what we are doing. Too few software people even try to use the machine on their own work. There are dozens of situations where a little machine computation would greatly aid the programmer. I recall one very simple one where a nonexpert with a very long FORTRAN program from the outside wanted to convert it to our local use, so he wrote a simple FORTRAN program to locate all the input-output statements and all the library references. In my experience, most programmers would have personally scanned long listings of the program to find them and with the usual human fallibility missed a couple the first time. I believe we need to convince the computer expert that the machine is his most powerful tool and that he should learn to use it as much as he can rather than personally scan the long listings of symbols as I see being done everywhere I go around the country. If what I am reporting is at all true, we have failed to teach this in the past. Of course some of the best people do in fact use the computer as I am recommending; my observation is that the run-of-the-mill programmers do not do so.

To parody our current methods of teaching programming, we give beginners a grammar and a dictionary and tell them that they are now great writers. We

seldom, if ever, give them any serious training in *style*. Indeed I have watched for years for the appearance of a *Manual of Style* and/or an *Anthology of Good Programming* and have as yet found none. Like writing, programming is a difficult and complex art. In both writing and programming, compactness is desirable but in both you can easily be too compact. When you consider how we teach good writing—the exercises, the compositions, and the talks that the student gives and is graded on by the teacher during his training in English—it seems we have been very remiss in this matter of teaching style in programming. Unfortunately only few programmers who admit that there is something in what I have called “style” are willing to formulate their feelings and to give specific examples. As a result, few programmers write in flowing poetry; most write in halting prose.

I doubt that style in programming is tied very closely to any particular machine or language, any more than good writing in one natural language is significantly different than it is in another. There are, of course, particular idioms and details in one language that favor one way of expressing the idea rather than another, but the essentials of good writing seem to transcend the differences in the Western European languages with which I am familiar. And I doubt that it is much different for most general purpose digital machines that are available these days.

Since I am apt to be misunderstood when I say we need more of an engineering flavor and less of a science one, I should perhaps point out that I came to computer science with a Ph.D. in pure mathematics. When I ask that the training in software be given a more practical, engineering flavor, I also loudly proclaim that we have too little understanding of what we are doing and that we desperately need to develop relevant theories.

Indeed, one of my major complaints about the computer field is that whereas Newton could say, “If I have seen a little farther than others it is because I have stood on the shoulders of giants,” I am forced to say, “Today we stand on each other’s feet.” Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things.

This brings me to another distinction, that between undirected research and basic research. Everyone likes to do undirected research and most people like to believe that undirected research is basic research. I am choosing to define basic research as being work upon which people will in the future base a lot of their work. After all, what else can we reasonably mean by basic research other than work upon which a lot of later work is based? I believe experience shows that relatively few people are *capable* of doing basic research. While one cannot be certain that a particular piece of work will or will not turn out to be basic, one can often give fairly accurate probabilities on the outcome. Upon examining the question of the nature of basic research, I have come to the conclusion that what determines whether or not a piece of work has much chance to become basic is not so much the question asked as it is the way the problem is attacked.

Numerical analysis is the one venerable part of our curriculum that is widely accepted as having some content. Yet all too often there is some justice in the remark that many of the textbooks are written for mathematicians and are in fact much more mathematics than they are practical computing. The reason is, of course, that many of the people in the field are converted, or rather only partially.

converted, mathematicians who still have the unconscious standards of mathematics in the back of their minds. I am sure many of you are familiar with my objections³ along these lines and I need not repeat them here.

It has been remarked to me by several persons, and I have also observed, that many of the courses in the proposed computer science curriculum are padded. Often they appear to cover every detail rather than confining themselves to the main ideas. We do not need to teach every method for finding the real zeros of a function: we need to teach a few typical ones that are both effective and illustrate basic concepts in numerical analysis. And what I have just said about numerical analysis goes even more for software courses. There do not seem to me (and to some others) to be enough fundamental ideas in all that we know of software to justify the large amount of time that is devoted to the topic. We should confine the material we teach to that which is important in ideas and technique—the plodding through a mass of minutiae should be avoided.

Let me now turn to the delicate matter of ethics. It has been observed on a number of occasions that the ethical behavior of the programmers in accounting installations leaves a lot to be desired when compared to that of the trained accounting personnel.⁴ We seem not to teach the "sacredness" of information about people and private company material. My limited observation of computer experts is that they have only the slightest regard for these matters. For example, most programmers believe they have the right to take with them any program they wish when they change employers. We should look at, and copy, how ethical standards are incorporated into the traditional accounting courses (and elsewhere), because they turn out a more ethical product than we do. We talk a lot in public of the dangers of large data banks of personnel records, but we do not do our share at the level of indoctrination of our own computer science majors.

Along these lines, let me briefly comment on the matter of professional standards. We have recently had a standard published⁵ and it seems to me to be a good one, but again I feel that I am justified in asking how this is being incorporated into the training of our students, how they are to learn to behave that way. Certainly it is not sufficient to read it to the class each morning; both ethical and professional behavior are not effectively taught that way. There is plenty of evidence that other professions do manage to communicate to their students professional standards which, while not always followed by every member, are certainly a lot better instilled than those we are presently providing for our students. Again, we need to examine how they do this kind of training and try to adapt their methods to our needs.

Lastly, let me mention briefly the often discussed topic of social responsibility. We have sessions at meetings on this topic, we discuss it in the halls and over coffee and beer, but again I ask, "How is it being incorporated into our training program?" The fact that we do not have exact rules to follow is not sufficient reason for omitting all training in this important matter.

I believe these three topics—ethics, professional behavior, and social responsibility—must be incorporated into the computer science curriculum. Personally

³ HAMMING, R. W. Numerical analysis vs. mathematics. *Science* 148 (Apr. 1965), 473-475.

⁴ CAREY, J. L., AND DOHERTY, W. A. Ethical Standards of the Accounting Profession. Am. Inst. CPAs., 1966.

⁵ Comm. ACM 11, 3 (Mar. 1968), 198-220.

I do not believe that a separate course on these topics will be effective. From what little I understand of the matter of teaching these kinds of things, they can best be taught by example, by the behavior of the professor. They are taught in the odd moments, by the way the professor phrases his remarks and handles himself. Thus it is the professor who must first be made conscious that a significant part of his teaching role is in communicating these delicate, elusive matters and that he is not justified in saying, "They are none of my business." These are things that must be taught *constantly, all the time, by everyone*, or they will not be taught at all. And if they are not somehow taught to the majority of our students, then the field will justly keep its present reputation (which may well surprise you if you ask your colleagues in other departments for their frank opinions).

In closing, let me revert to a reasonable perspective of the computer science field. The field is very new, it has had to run constantly just to keep up, and there has been little time for many of the things we have long known we must some day do. But at least in the universities we have finally arrived: we have established separate departments with reasonable courses, faculty, and equipment. We are now well started, and it is time to deepen, strengthen, and improve our field so that we can be justly proud of what we teach, how we teach it, and of the students we turn out. We are not engaged in turning out technicians, idiot savants, and computniks; we know that in this modern, complex world we must turn out people who can play responsible major roles in our changing society, or else we must acknowledge that we have failed in our duty as teachers and leaders in this exciting, important field—computer science.

**1972 ACM Turing
Award Lecture**

[Extract from the *Turing Award Citation* read by M.D. McIlroy, chairman of the ACM Turing Award Committee, at the presentation of this lecture on August 14, 1972, at the ACM Annual Conference in Boston.]

The working vocabulary of programmers everywhere is studded with words originated or forcefully promulgated by E.W. Dijkstra—display, deadly embrace, semaphore, go-to-less programming, structured programming. But his influence on programming is more pervasive than any

glossary can possibly indicate. The precious gift that this Turing Award acknowledges is Dijkstra's *style*: his approach to programming as a high, intellectual challenge; his eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; and his illuminating perception of problems at the foundations of program design. He has published about a dozen papers, both technical and reflective, among which are especially to be noted his philo-

sophical addresses at IFIP,¹ his already classic papers on cooperating sequential processes,² and his memorable indictment of the go-to statement.³ An influential series of letters by Dijkstra have recently surfaced as a polished monograph on the art of composing programs.⁴

We have come to value good programs in much the same way as we value good literature. And at the center of this movement, creating and reflecting patterns no less beautiful than useful, stands E.W. Dijkstra.

The Humble Programmer

by Edsger W. Dijkstra



As a result of a long sequence of coincidences I entered the programming profession officially on the first spring morning of 1952, and as far as I have been able to trace, I was the first Dutchman to do so in my country. In retrospect the most amazing thing is the slowness with which, at least in my part of the world, the programming profession emerged, a slowness which is now hard to believe. But I am grateful for two vivid recollections from that period that establish that slowness beyond any doubt.

After having programmed for some three years, I had a discussion with van Wijngaarden, who was then my boss at the Mathematical Centre in Amsterdam—a discussion for which I shall remain grateful to him as long as I live. The point was that

I was supposed to study theoretical physics at the University of Leiden simultaneously, and as I found the two activities harder and harder to combine, I had to make up my mind, either to stop programming and become a real, respectable theoretical physicist, or to carry my study of physics to a formal completion only, with a minimum of effort, and to become . . . , yes what? A programmer? But was that a respectable profession? After all, what was programming? Where was the sound body of knowledge that could sup-

Copyright © 1972, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

^{1,2,3,4} Footnotes are on page 866.

port it as an intellectually respectable discipline? I remember quite vividly how I envied my hardware colleagues, who, when asked about their professional competence, could at least point out that they knew everything about vacuum tubes, amplifiers and the rest, whereas I felt that, when faced with that question, I would stand empty-handed. Full of misgivings I knocked on van Wijnagaarden's office door, asking him whether I could speak to him for a moment; when I left his office a number of hours later, I was another person. For after having listened to my problems patiently, he agreed that up till that moment there was not much of a programming discipline, but then he went on to explain quietly that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come? This was a turning point in my life and I completed my study of physics formally as quickly as I could. One moral of the above story is, of course, that we must be very careful when we give advice to younger people: sometimes they follow it!

Two years later, in 1957, I married, and Dutch marriage rites require you to state your profession and I stated that I was a programmer. But the municipal authorities of the town of Amsterdam did not accept it on the grounds that there was no such profession. And, believe it or not, but under the heading "profession" my marriage record shows the ridiculous entry "theoretical physicist"!

So much for the slowness with which I saw the programming profession emerge in my own country. Since then I have seen more of the world, and it is my general impression that in other countries, apart from a possible shift of dates, the growth pattern has been very much the same.

Let me try to capture the situation in those old days in a little bit more detail, in the hope of getting a better understanding of the situa-

tion today. While we pursue our analysis, we shall see how many common misunderstandings about the true nature of the programming task can be traced back to that now distant past.

The first automatic electronic computers were all unique, single-copy machines and they were all to be found in an environment with the exciting flavor of an experimental laboratory. Once the vision of the automatic computer was there, its realization was a tremendous challenge to the electronic technology then available, and one thing is certain: we cannot deny the courage of the groups that decided to try to build such a fantastic piece of equipment. For fantastic pieces of equipment they were: in retrospect one can only wonder that those first machines worked at all, at least sometimes. The overwhelming problem was to get and keep the machine in working order. The preoccupation with the physical aspects of automatic computing is still reflected in the names of the older scientific societies in the field, such as the Association for Computing Machinery or the British Computer Society, names in which explicit reference is made to the physical equipment.

What about the poor programmer? Well, to tell the honest truth, he was hardly noticed. For one thing, the first machines were so bulky that you could hardly move them and besides that, they required such extensive maintenance that it was quite natural that the place where people tried to use the machine was the same laboratory where the machine had been developed. Secondly, the programmer's somewhat invisible work was without any glamour: you could show the machine to visitors and that was several orders of magnitude more spectacular than some sheets of coding. But most important of all, the programmer himself had a very modest view of his own work: his work derived all its significance from the existence of that wonderful machine. Because that was a unique machine, he knew only too well that his programs had only local signifi-

cance, and also because it was palpably obvious that this machine would have a limited lifetime, he knew that very little of his work would have a lasting value. Finally, there is yet another circumstance that had a profound influence on the programmer's attitude toward his work: on the one hand, besides being unreliable, his machine was usually too slow and its memory was usually too small, i.e. he was faced with a pinching shoe, while on the other hand its usually somewhat queer order code would cater for the most unexpected constructions. And in those days many a clever programmer derived an immense intellectual satisfaction from the cunning tricks by means of which he contrived to squeeze the impossible into the constraints of his equipment.

Two opinions about programming date from those days. I mention them now; I shall return to them later. The one opinion was that a really competent programmer should be puzzle-minded and very fond of clever tricks; the other opinion was that programming was nothing more than optimizing the efficiency of the computational process, in one direction or the other.

The latter opinion was the result of the frequent circumstance that, indeed, the available equipment was a painfully pinching shoe, and in those days one often encountered the naive expectation that, once more powerful machines were available, programming would no longer be a problem, for then the struggle to push the machine to its limits would no longer be necessary and that was all that programming was about, wasn't it? But in the next decades something completely different happened: more powerful machines became available, not just an order of magnitude more powerful, even several orders of magnitude more powerful. But instead of finding ourselves in a state of eternal bliss with all programming problems solved, we found ourselves up to our necks in the software crisis! How come?

There is a minor cause: in one or two respects modern machinery

is basically more difficult to handle than the old machinery. Firstly, we have got the I/O interrupts, occurring at unpredictable and irreproducible moments; compared with the old sequential machine that pretended to be a fully deterministic automaton, this has been a dramatic change, and many a systems programmer's grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature. Secondly, we have got machines equipped with multilevel stores, presenting us problems of management strategy that, in spite of the extensive literature on the subject, still remain rather elusive. So much for the added complication due to structural changes of the actual machines.

But I called this a minor cause; the major cause is . . . that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them—it has created the problem of using its products. To put it in another way: as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means. The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he *had* to dream about them and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis? No, certainly not, and as you may guess, it was even predicted well in advance; but the

trouble with minor prophets, of course, is that it is only five years later that you really know that they had been right.

Then, in the mid sixties something terrible happened: the computers of the so-called third generation made their appearance. The official literature tells us that their price/performance ratio has been one of the major design objectives. But if you take as "performance" the duty cycle of the machine's various components, little will prevent you from ending up with a design in which the major part of your performance goal is reached by internal housekeeping activities of doubtful necessity. And if your definition of price is the price to be paid for the hardware, little will prevent you from ending up with a design that is terribly hard to program for: for instance the order code might be such as to enforce, either upon the programmer or upon the system, early binding decisions presenting conflicts that really cannot be resolved. And to a large extent these unpleasant possibilities seem to have become reality.

When these machines were announced and their functional specifications became known, many among us must have become quite miserable; at least I was. It was only reasonable to expect that such machines would flood the computing community, and it was therefore all the more important that their design should be as sound as possible. But the design embodied such serious flaws that I felt that with a single stroke the progress of computing science had been retarded by at least ten years; it was then that I had the blackest week in the whole of my professional life. Perhaps the most saddening thing now is that, even after all those years of frustrating experience, still so many people honestly believe that some law of nature tells us that machines have to be that way. They silence their doubts by observing how many of these machines have been sold, and derive from that observation the false sense of security that, after all, the

design cannot have been that bad. But upon closer inspection, that line of defense has the same convincing strength as the argument that cigarette smoking must be healthy because so many people do it.

It is in this connection that I regret that it is not customary for scientific journals in the computing area to publish reviews of newly announced computers in much the same way as we review scientific publications; to review machines would be at least as important. And here I have a confession to make: in the early sixties I wrote such a review with the intention of submitting it to Communications, but in spite of the fact that the few colleagues to whom the text was sent for their advice urged me to do so, I did not dare to do it, fearing that the difficulties either for myself or for the Editorial Board would prove to be too great. This suppression was an act of cowardice on my side for which I blame myself more and more. The difficulties I foresaw were a consequence of the absence of generally accepted criteria, and although I was convinced of the validity of the criteria I had chosen to apply, I feared that my review would be refused or discarded as "a matter of personal taste." I still think that such reviews would be extremely useful and I am longing to see them appear, for their accepted appearance would be a sure sign of maturity of the computing community.

The reason that I have paid the above attention to the hardware scene is because I have the feeling that one of the most important aspects of any computing tool is its influence on the thinking habits of those who try to use it, and because I have reasons to believe that that influence is many times stronger than is commonly assumed. Let us now switch our attention to the software scene.

Here the diversity has been so large that I must confine myself to a few stepping stones. I am painfully aware of the arbitrariness of my choice, and I beg you not to draw any conclusions with regard to my appreciation of the many efforts that

will have to remain unmentioned.

In the beginning there was the EDSAC in Cambridge, England, and I think it quite impressive that right from the start the notion of a subroutine library played a central role in the design of that machine and of the way in which it should be used. It is now nearly 25 years later and the computing scene has changed dramatically, but the notion of basic software is still with us, and the notion of the closed subroutine is still one of the key concepts in programming. We should recognize the closed subroutine as one of the greatest software inventions; it has survived three generations of computers and it will survive a few more, because it caters for the implementation of one of our basic patterns of abstraction. Regrettably enough, its importance has been underestimated in the design of the third generation computers, in which the great number of explicitly named registers of the arithmetic unit implies a large overhead on the subroutine mechanism. But even that did not kill the concept of the subroutine, and we can only pray that the mutation won't prove to be hereditary.

The second major development on the software scene that I would like to mention is the birth of FORTRAN. At that time this was a project of great temerity, and the people responsible for it deserve our great admiration. It would be absolutely unfair to blame them for shortcomings that only became apparent after a decade or so of extensive usage; groups with a successful lookahead of ten years are quite rare! In retrospect we must rate FORTRAN as a successful coding technique, but with very few effective aids to conception, aids which are now so urgently needed that time has come to consider it out of date. The sooner we can forget that FORTRAN ever existed, the better, for as a vehicle of thought it is no longer adequate: it wastes our brainpower, and it is too risky and therefore too expensive to use. FORTRAN's tragic fate has been its wide acceptance, mentally chaining thousands and thousands of pro-

grammers to our past mistakes. I pray daily that more of my fellow-programmers may find the means of freeing themselves from the curse of compatibility.

The third project I would not like to leave unmentioned is LISP, a fascinating enterprise of a completely different nature. With a few very basic principles at its foundation, it has shown a remarkable stability. Besides that, LISP has been the carrier for a considerable number of, in a sense, our most sophisticated computer applications. LISP has jokingly been described as "the most intelligent way to misuse a computer." I think that description a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.

The fourth project to be mentioned is ALGOL 60. While up to the present day FORTRAN programmers still tend to understand their programming language in terms of the specific implementation they are working with—hence the prevalence of octal or hexadecimal dumps—while the definition of LISP is still a curious mixture of what the language means and how the mechanism works, the famous Report on the Algorithmic Language ALGOL 60 is the fruit of a genuine effort to carry abstraction a vital step further and to define a programming language in an implementation-independent way. One could argue that in this respect its authors have been so successful that they have created serious doubts as to whether it could be implemented at all! The report gloriously demonstrated the power of the formal method BNF, now fairly known as Backus-Naur-Form, and the power of carefully phrased English, at least when used by someone as brilliant as Peter Naur. I think that it is fair to say that only very few documents as short as this have had an equally profound influence on the computing community. The ease with which in later years the names ALGOL and ALGOL-like have been used, as an unpro-

tected trademark, to lend glory to a number of sometimes hardly related younger projects is a somewhat shocking compliment to ALGOL's standing. The strength of BNF as a defining device is responsible for what I regard as one of the weaknesses of the language: an overelaborate and not too systematic syntax could now be crammed into the confines of very few pages. With a device as powerful as BNF, the Report on the Algorithmic Language ALGOL 60 should have been much shorter. Besides that, I am getting very doubtful about ALGOL 60's parameter mechanism: it allows the programmer so much combinatorial freedom that its confident use requires a strong discipline from the programmer. Besides being expensive to implement, it seems dangerous to use.

Finally, although the subject is not a pleasant one, I must mention PL/I, a programming language for which the defining documentation is of a frightening size and complexity. Using PL/I must be like flying a plane with 7,000 buttons, switches, and handles to manipulate in the cockpit. I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language—our basic tool, mind you!—already escapes our intellectual control. And if I have to describe the influence PL/I can have on its users, the closest metaphor that comes to my mind is that of a drug. I remember from a symposium on higher level programming languages a lecture given in defense of PL/I by a man who described himself as one of its devoted users. But within a one-hour lecture in praise of PL/I, he managed to ask for the addition of about 50 new "features," little supposing that the main source of his problems could very well be that it contained already far too many "features." The speaker displayed all the depressing symptoms of addiction, reduced as he was to the state of mental stagnation in which he could only ask for more, more, more....

When FORTRAN has been called an infantile disorder, full PL/I, with its growth characteristics of a dangerous tumor, could turn out to be a fatal disease.

So much for the past. But there is no point in making mistakes unless thereafter we are able to learn from them. As a matter of fact, I think that we have learned so much that within a few years programming can be an activity vastly different from what it has been up till now, so different that we had better prepare ourselves for the shock. Let me sketch for you one of the possible futures. At first sight, this vision of programming in perhaps already the near future may strike you as utterly fantastic. Let me therefore also add the considerations that might lead one to the conclusion that this vision could be a very real possibility.

The vision is that, well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs. These two improvements go hand in hand. In the latter respect software seems to be different from many other products, where as a rule a higher quality implies a higher price. Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging—they should not introduce the bugs to start with. In other words, both goals point to the same change.

Such a drastic change in such a short period of time would be a revolution, and to all persons that base their expectations for the future on smooth extrapolation of the recent past—appealing to some unwritten laws of social and cultural inertia—the chance that this drastic

change will take place must seem negligible. But we all know that sometimes revolutions do take place! And what are the chances for this one?

There seem to be three major conditions that must be fulfilled. The world at large must recognize the need for the change; secondly, the economic need for it must be sufficiently strong; and, thirdly, the change must be technically feasible. Let me discuss these three conditions in the above order.

With respect to the recognition of the need for greater reliability of software, I expect no disagreement anymore. Only a few years ago this was different: to talk about a software crisis was blasphemy. The turning point was the Conference on Software Engineering in Garmisch, October 1968, a conference that created a sensation as there occurred the first open admission of the software crisis. And by now it is generally recognized that the design of any large sophisticated system is going to be a very difficult job, and whenever one meets people responsible for such undertakings, one finds them very much concerned about the reliability issue, and rightly so. In short, our first condition seems to be satisfied.

Now for the economic need. Nowadays one often encounters the opinion that in the sixties programming has been an overpaid profession, and that in the coming years programmer salaries may be expected to go down. Usually this opinion is expressed in connection with the recession, but it could be a symptom of something different and quite healthy, *viz.* that perhaps the programmers of the past decade have not done so good a job as they should have done. Society is getting dissatisfied with the performance of programmers and of their products. But there is another factor of much greater weight. In the present situation it is quite usual that for a specific system, the price to be paid for the development of the software is of the same order of magnitude as the price of the hardware needed,

and society more or less accepts that. But hardware manufacturers tell us that in the next decade hardware prices can be expected to drop with a factor of ten. If software development were to continue to be the same clumsy and expensive process as it is now, things would get completely out of balance. You cannot expect society to accept this, and therefore we *must* learn to program an order of magnitude more effectively. To put it in another way: as long as machines were the largest item on the budget, the programming profession could get away with its clumsy techniques; but that umbrella will fold very rapidly. In short, also our second condition seems to be satisfied.

And now the third condition: is it technically feasible? I think it might be, and I shall give you six arguments in support of that opinion.

A study of program structure has revealed that programs—even alternative programs for the same task and with the same mathematical content—can differ tremendously in their intellectual manageability. A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program. These rules are of two kinds. Those of the first kind are easily imposed mechanically, *viz.* by a suitably chosen programming language. Examples are the exclusion of goto-statements and of procedures with more than one output parameter. For those of the second kind, I at least—but that may be due to lack of competence on my side—see no way of imposing them mechanically, as it seems to need some sort of automatic theorem prover for which I have no existence proof. Therefore, for the time being and perhaps forever, the rules of the second kind present themselves as elements of discipline required from the programmer. Some of the rules I have in mind are so clear that they can be taught and that there never needs to be an argument as to whether a given program violates them or not. Examples are the re-

uirements that no loop should be written down without providing a proof for termination or without stating the relation whose invariance will not be destroyed by the execution of the repeatable statement.

I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs. If someone fears that this restriction is so severe that we cannot live with it, I can reassure him: the class of intellectually manageable programs is still sufficiently rich to contain many very realistic programs for any problem capable of algorithmic solution. We must not forget that it is *not* our business to make programs; it is our business to design classes of computations that will display a desired behavior. The suggestion of confining ourselves to intellectually manageable programs is the basis for the first two of my announced six arguments.

Argument one is that, as the programmer only needs to consider intellectually manageable programs, the alternatives he is choosing from are much, much easier to cope with.

Argument two is that, as soon as we have decided to restrict ourselves to the subset of the intellectually manageable programs, we have achieved, once and for all, a drastic reduction of the solution space to be considered. And this argument is distinct from argument one.

Argument three is based on the constructive approach to the problem of program correctness. Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program

grow hand in hand. Argument three is essentially based on the following observation. If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance. By definition this approach is only applicable when we restrict ourselves to intellectually manageable programs, but it provides us with effective means for finding a satisfactory one among these.

Argument four has to do with the way in which the amount of intellectual effort needed to design a program depends on the program length. It has been suggested that there is some law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. And this is because it need not be true. We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called "abstraction"; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer. In this connection it might be worthwhile to point out that the purpose of abstracting is *not* to be vague, but to create a new semantic level in which one can be absolutely precise. Of course I have tried to find a fundamental cause that would prevent our abstraction mechanisms from being sufficiently effective. But no matter how hard I tried, I did not find such a cause. As a result I tend to the assumption—up till now not disproved by experience—that by suitable application of our powers of abstraction, the intellectual effort required to conceive or to understand a program need not grow more than proportional to program length. A by-product of these investigations may be of much greater practical significance, and is, in fact, the basis of my fourth argument. The by-product was the identification of a number

of patterns of abstraction that play a vital role in the whole process of composing programs. Enough is known about these patterns of abstraction that you could devote a lecture to each of them. What the familiarity and conscious knowledge of these patterns of abstraction imply dawned upon me when I realized that, had they been common knowledge 15 years ago, the step from BNF to syntax-directed compilers, for instance, could have taken a few minutes instead of a few years. Therefore I present our recent knowledge of vital abstraction patterns as the fourth argument.

Now for the fifth argument. It has to do with the influence of the tool we are trying to use upon our own thinking habits. I observe a cultural tradition, which in all probability has its roots in the Renaissance, to ignore this influence, to regard the human mind as the supreme and autonomous master of its artifacts. But if I start to analyze the thinking habits of myself and of my fellow human beings, I come, whether I like it or not, to a completely different conclusion, *viz.* that the tools we are trying to use and the language or notation we are using to express or record our thoughts are the major factors determining what we can think or express at all! The analysis of the influence that programming languages have on the thinking habits of their users, and the recognition that, by now, brain-power is by far our scarcest resource, these together give us a new collection of yardsticks for comparing the relative merits of various programming languages. The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. In the case of a well-known conversational programming language I have been told from various sides that as soon as a programming community is equipped with a terminal for it, a specific phenomenon occurs that even has a well-established name: it is

called "the one-liners." It takes one of two different forms: one programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question, "Can you code this in less symbols?"—as if this were of any conceptual relevance!—or he just says, "Guess what it does!" From this observation we must conclude that this language as a tool is an open invitation for clever tricks; and while exactly this may be the explanation for some of its appeal, *viz.* to those who like to show how clever they are, I am sorry, but I must regard this as one of the most damning things that can be said about a programming language. Another lesson we should have learned from the recent past is that the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages. When I say "modest," I mean that, for instance, not only ALGOL 60's "for clause," but even FORTRAN's "DO loop" may find themselves thrown out as being too baroque. I have run a little programming experiment with really experienced volunteers, but something quite unintended and quite unexpected turned up. None of my volunteers found the obvious and most elegant solution. Upon closer analysis this turned out to have a common source: their notion of repetition was so tightly connected to the idea of an associated controlled variable to be stepped up, that they were mentally blocked from seeing the obvious. Their solutions were less efficient, needlessly hard to understand, and it took them a very long time to find them. It was a revealing, but also shocking experience for me. Finally, in one respect one hopes that tomorrow's programming languages will differ greatly from what we are used to now: to a much greater extent than hitherto they should invite us to reflect in

the structure of what we write down all abstractions needed to cope conceptually with the complexity of what we are designing. So much for the greater adequacy of our future tools, which was the basis of the fifth argument.

As an aside I would like to insert a warning to those who identify the difficulty of the programming task with the struggle against the inadequacies of our current tools, because they might conclude that, once our tools will be much more adequate, programming will no longer be a problem. Programming will remain very difficult, because once we have freed ourselves from the circumstantial clumsiness, we will find ourselves free to tackle the problems that are now well beyond our programming capacity.

You can quarrel with my sixth argument, for it is not so easy to collect experimental evidence for its support, a fact that will not prevent me from believing in its validity. Up till now I have not mentioned the word "hierarchy," but I think that it is fair to say that this is a key concept for all systems embodying a nicely factored solution. I could even go one step further and make an article of faith out of it, *viz.* that the only problems we can really solve in a satisfactory manner are those that finally admit a nicely factored solution. At first sight this view of human limitations may strike you as a rather depressing view of our predicament, but I don't feel it that way. On the contrary, the best way to learn to live with our limitations is to know them. By the time that we are sufficiently modest to try factored solutions only, because the other efforts escape our intellectual grip, we shall do our utmost to avoid all those interfaces impairing our ability to factor the system in a helpful way. And I can not but expect that this will repeatedly lead to the discovery that an initially untractable problem can be factored after all. Anyone who has seen how the majority of the troubles of the compiling phase called "code generation" can be tracked down to funny prop-

erties of the order code will know a simple example of the kind of things I have in mind. The wider applicability of nicely factored solutions is my sixth and last argument for the technical feasibility of the revolution that might take place in the current decade.

In principle I leave it to you to decide for yourself how much weight you are going to give to my considerations, knowing only too well that I can force no one else to share my beliefs. As in each serious revolution, it will provoke violent opposition and one can ask oneself where to expect the conservative forces trying to counteract such a development. I don't expect them primarily in big business, not even in the computer business; I expect them rather in the educational institutions that provide today's training and in those conservative groups of computer users that think their old programs so important that they don't think it worthwhile to rewrite and improve them. In this connection it is sad to observe that on many a university campus the choice of the central computing facility has too often been determined by the demands of a few established but expensive applications with a disregard of the question, how many thousands of "small users" who are willing to write their own programs are going to suffer from this choice. Too often, for instance, high-energy physics seems to have blackmailed the scientific community with the price of its remaining experimental equipment. The easiest answer, of course, is a flat denial of the technical feasibility, but I am afraid that you need pretty strong arguments for that. No reassurance, alas, can be obtained from the remark that the intellectual ceiling of today's average programmer will prevent the revolution from taking place: with others programming so much more effectively, he is liable to be edged out of the picture anyway.

There may also be political impediments. Even if we know how to educate tomorrow's professional programmer, it is not certain that

the society we are living in will allow us to do so. The first effect of teaching a methodology—rather than disseminating knowledge—is that of enhancing the capacities of the already capable, thus magnifying the difference in intelligence. In a society in which the educational system is used as an instrument for the establishment of a homogenized culture, in which the cream is prevented from rising to the top, the education of competent programmers could be politically unpalatable.

Let me conclude. Automatic computers have now been with us for a quarter of a century. They have had a great impact on our society in their capacity of tools, but in that capacity their influence will be but a ripple on the surface of our culture compared with the much more profound influence they will have in their capacity of intellectual challenge which will be without precedent in the cultural history of mankind. Hierarchical systems seem to have the property that something considered as an undivided entity on one level is considered as a composite object on the next lower level of greater detail; as a result the natural grain of space or time that is applicable at each level decreases by an order of magnitude when we shift our attention from one level to the next lower one. We understand walls in terms of bricks, bricks in terms of crystals, crystals in terms of molecules, etc. As a result the number of levels that can be distinguished meaningfully in a hierarchical system is kind of proportional to the logarithm of the ratio between the largest and the smallest grain, and therefore, unless this ratio is very large, we cannot expect many levels. In computer programming our basic building block has an associated time grain of less than a microsecond, but our program may take hours of

computation time. I do not know of any other technology covering a ratio of 10^{10} or more: the computer, by virtue of its fantastic speed, seems to be the first to provide us with an environment where highly hierarchical artifacts are both possible and necessary. This challenge, *viz.* the confrontation with the programming task, is so unique that this novel experience can teach us a lot about ourselves. It should deepen our understanding of the processes of design and creation; it should give us better control over the task of organizing our thoughts. If it did not do so, to my taste we should not deserve the computer at all!

It has already taught us a few lessons, and the one I have chosen to stress in this talk is the following. We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as *Very Humble Programmers*.

[References to the following footnotes are found in the extract from the Turing Award citation on page 859.]

¹Some meditations on advanced programming, Proceedings of the IFIP Congress 1962, 535-538; Programming considered as a human activity, Proceedings of the IFIP Congress 1965, 213-217.

²Solution of a problem in concurrent pro-

gramming, control, CACM 8 (Sept. 1965), 569; The structure of the "THE" multiprogramming system, CACM 11 (May, 1968), 341-346.

³Go to statement considered harmful, CACM 11 (Mar. 1968), 147-148.

⁴A short introduction to the art of computer programming, Technische Hogeschool, Eindhoven, 1971.

1979 ACM Turing Award Lecture

Delivered at ACM '79, Detroit, Oct. 29, 1979

The 1979 ACM Turing Award was presented to Kenneth E. Iverson by Walter Carlson, Chairman of the Awards Committee, at the ACM Annual Conference in Detroit, Michigan, October 29, 1979.

In making its selection, the General Technical Achievement Award Committee cited Iverson for his pioneering effort in programming languages and mathematical notation resulting in what the computing field now knows as APL. Iverson's contributions to the implementation of interactive systems, to the educational uses of APL, and to programming language theory and practice were also noted.

Born and raised in Canada, Iverson received his doctorate in 1954 from Harvard University. There he served as Assistant Professor of Applied Mathematics from 1955-1960. He then joined International Business Machines, Corp. and in 1970 was named an IBM Fellow in honor of his contribution to the development of APL.

Dr. Iverson is presently with I.P. Sharp Associates in Toronto. He has published numerous articles on programming languages and has written four books about programming and mathematics: *A Programming Language* (1962), *Elementary Functions* (1966), *Algebra: An Algorithmic Treatment* (1972), and *Elementary Analysis* (1976).

Notation as a Tool of Thought

Kenneth E. Iverson
IBM Thomas J. Watson Research Center



Key Words and Phrases: APL, mathematical notation

CR Category: 4.2

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Author's present address: K.E. Iverson, I.P. Sharp Associates, 145 King Street West, Toronto, Ontario, Canada M5H1J8.
© 1980 ACM 0001-0782/80/0800-0444 \$00.75.

The importance of nomenclature, notation, and language as tools of thought has long been recognized. In chemistry and in botany, for example, the establishment of systems of nomenclature by Lavoisier and Linnaeus did much to stimulate and to channel later investigation. Concerning language, George Boole in his *Laws of Thought* [1, p.24] asserted "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted."

Mathematical notation provides perhaps the best-known and best-developed example of language used consciously as a tool of thought. Recognition of the important role of notation in mathematics is clear from the quotations from mathematicians given in Cajori's *A History of Mathematical Notations* [2, pp.332,331]. They are well worth reading in full, but the following excerpts suggest the tone:

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.

A.N. Whitehead

The quantity of meaning compressed into small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid.

Charles Babbage

Nevertheless, mathematical notation has serious deficiencies. In particular, it lacks universality, and must be interpreted differently according to the topic, according to the author, and even according to the immediate context. Programming languages, because they were designed for the purpose of directing computers, offer important advantages as tools of thought. Not only are they universal (general-purpose), but they are also executable and unambiguous. Executability makes it possible to use computers to perform extensive experiments on ideas expressed in a programming language, and the lack of ambiguity makes possible precise thought experiments. In other respects, however, most programming languages are decidedly inferior to mathematical notation and are little used as tools of thought in ways that would be considered significant by, say, an applied mathematician.

The thesis of the present paper is that the advantages of executability and universality found in programming languages can be effectively combined, in a single coherent language, with the advantages offered by mathematical notation. It is developed in four stages:

- (a) Section 1 identifies salient characteristics of mathematical notation and uses simple problems to illustrate how these characteristics may be provided in an executable notation.
- (b) Sections 2 and 3 continue this illustration by deeper treatment of a set of topics chosen for their general interest and utility. Section 2 concerns polynomials, and Section 3 concerns transformations between representations of functions relevant to a number of topics, including permutations and directed graphs. Although these topics might be characterized as mathematical, they are directly relevant to computer programming, and their relevance will increase as programming continues to develop into a legitimate mathematical discipline.
- (c) Section 4 provides examples of identities and formal proofs. Many of these formal proofs concern identities established informally and used in preceding sections.
- (d) The concluding section provides some general comparisons with mathematical notation, references to treatments of other topics, and discussion of the problem of introducing notation in context.

The executable language to be used is APL, a general purpose language which originated in an attempt to provide clear and precise expression in writing and teaching, and which was implemented as a programming language only after several years of use and development [3].

Although many readers will be unfamiliar with APL, I have chosen not to provide a separate introduction to it, but rather to introduce it in context as needed. Mathematical notation is always introduced in this way rather than being taught, as programming languages commonly are, in a separate course. Notation suited as a tool of thought in any topic should permit easy introduction in the context of that topic; one advantage of introducing APL in context here is that the reader may assess the relative difficulty of such introduction.

However, introduction in context is incompatible with complete discussion of all nuances of each bit of notation, and the reader must be prepared to either extend the definitions in obvious and systematic ways as required in later uses, or to consult a reference work. All of the notation used here is summarized in Appendix A, and is covered fully in pages 24-60 of *APL Language* [4].

Readers having access to some machine embodiment of APL may wish to translate the function definitions given here in *direct definition* form [5, p.10] (using « and » to represent the left and right arguments) to the *canonical* form required for execution. A function for performing this translation automatically is given in Appendix B.

1. Important Characteristics of Notation

In addition to the executability and universality emphasized in the introduction, a good notation should embody characteristics familiar to any user of mathematical notation:

- Ease of expressing constructs arising in problems.
- Suggestivity.
- Ability to subordinate detail.
- Economy.
- Amenity to formal proofs.

The foregoing is not intended as an exhaustive list, but will be used to shape the subsequent discussion.

Unambiguous executability of the notation introduced remains important, and will be emphasized by displaying below an expression the explicit result produced by it. To maintain the distinction between expressions and results, the expressions will be indented as they automatically are on APL computers. For example, the *integer* function denoted by `i` produces a vector of the first n integers

when applied to the argument n , and the *sum reduction* denoted by \leftrightarrow produces the sum of the elements of its vector argument, and will be shown as follows:

$$\begin{array}{c} 15 \\ 1 \ 2 \ 3 \ 4 \ 5 \\ +/15 \\ 15 \end{array}$$

We will use one non-executable bit of notation: the symbol \leftrightarrow appearing between two expressions asserts their equivalence.

1.1 Ease of Expressing Constructs Arising in Problems

If it is to be effective as a tool of thought, a notation must allow convenient expression not only of notions arising directly from a problem, but also of those arising in subsequent analysis, generalization, and specialization.

Consider, for example, the crystal structure illustrated by Figure 1, in which successive layers of atoms lie not directly on top of one another, but lie "close-packed" between those below them. The numbers of atoms in successive rows from the top in Figure 1 are therefore given by 15 , and the total number is given by $+/15$.

The three-dimensional structure of such a crystal is also close-packed; the atoms in the plane lying above Figure 1 would lie between the atoms in the plane below it, and would have a base row of four atoms. The complete three-dimensional structure corresponding to Figure 1 is therefore a tetrahedron whose planes have bases of lengths 1, 2, 3, 4, and 5. The numbers in successive planes are therefore the *partial sums* of the vector 15 , that is, the sum of the first element, the sum of the first two elements, etc. Such partial sums of a vector v are denoted by $+v$, the function $+v$ being called *sum scan*. Thus:

$$\begin{array}{c} +v15 \\ 1 \ 3 \ 6 \ 10 \ 15 \\ +/+v15 \\ 35 \end{array}$$

The final expression gives the total number of atoms in the tetrahedron.

The sum $+/15$ can be represented graphically in other ways, such as shown on the left of Figure 2. Combined with the inverted pattern on the right, this representation suggests that the sum may be simply related to the number of units in a rectangle, that is, to a product.

The lengths of the rows of the figure formed by pushing together the two parts of Figure 2 are given by adding the vector 15 to the same vector reversed. Thus:

$$\begin{array}{c} 15 \\ 1 \ 2 \ 3 \ 4 \ 5 \\ \phi15 \\ 5 \ 4 \ 3 \ 2 \ 1 \\ (15) + (\phi15) \\ 6 \ 6 \ 6 \ 6 \ 6 \end{array}$$

Fig. 1.

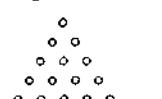


Fig. 2.



This pattern of 5 repetitions of 6 may be expressed as $5p6$, and we have:

$$\begin{array}{c} 5p6 \\ 6 \ 6 \ 6 \ 6 \ 6 \\ +/5p6 \\ 30 \\ 6 \times 5 \\ 30 \end{array}$$

The fact that $+/5p6 \leftrightarrow 6 \times 5$ follows from the definition of multiplication as repeated addition.

The foregoing suggests that $+/15 \leftrightarrow (6 \times 5) + 2$, and, more generally, that:

$$+/(N \leftrightarrow ((N+1) \times N) + 2)$$

A.1

1.2 Suggestivity

A notation will be said to be *suggestive* if the forms of the expressions arising in one set of problems suggest related expressions which find application in other problems. We will now consider related uses of the functions introduced thus far, namely:

$$+ \ \Phi \ \rho \ \leftrightarrow \ +/\ \leftrightarrow \ +\backslash$$

The example:

$$\begin{array}{c} 5p2 \\ 2 \ 2 \ 2 \ 2 \ 2 \\ \times/5p2 \\ 32 \end{array}$$

suggests that $\times/MpN \leftrightarrow N^M$, where \times represents the power function. The similarity between the definitions of power in terms of times, and of times in terms of plus may therefore be exhibited as follows:

$$\begin{array}{c} \times/MpN \leftrightarrow N^M \\ +/MpN \leftrightarrow N \times M \end{array}$$

Similar expressions for partial sums and partial products may be developed as follows:

$$\begin{array}{c} +\backslash 5p2 \\ 2 \ 4 \ 8 \ 16 \ 32 \\ 2 \times 15 \\ 2 \ 4 \ 8 \ 16 \ 32 \\ \times\backslash MpN \leftrightarrow N^{1/M} \\ +\backslash MpN \leftrightarrow N \times 1/M \end{array}$$

Because they can be represented by a triangle as in Figure 1, the sums $+v15$ are called *triangular numbers*. They are a special case of the *figurate numbers* obtained by repeated applications of sum scan, beginning either with $+v1n$, or with $+vMp1$. Thus:

$$\begin{array}{c} 5p1 \\ 1 \ 1 \ 1 \ 1 \ 1 \\ +\backslash +\backslash 5p1 \\ 1 \ 3 \ 6 \ 10 \ 15 \\ +\backslash 5p1 \\ 1 \ 2 \ 3 \ 4 \ 5 \\ +\backslash +\backslash +\backslash 5p1 \\ 1 \ 4 \ 10 \ 20 \ 35 \end{array}$$

Replacing sums over the successive integers by products yields the factorials as follows:

$$\begin{array}{r} \begin{array}{c} 15 \\ 1 \ 2 \ 3 \ 4 \ 5 \\ \times / 15 \\ 120 \\ \quad : 5 \\ 120 \end{array} & \begin{array}{c} \times \backslash 15 \\ 1 \ 2 \ 6 \ 24 \ 120 \\ \quad : 15 \\ 1 \ 2 \ 6 \ 24 \ 120 \end{array} \end{array}$$

Part of the suggestive power of a language resides in the ability to represent identities in brief, general, and easily remembered forms. We will illustrate this by expressing *dualities* between functions in a form which embraces DeMorgan's laws, multiplication by the use of logarithms, and other less familiar identities.

If v is a vector of positive numbers, then the product \times/v may be obtained by taking the natural logarithms of each element of v (denoted by $\bullet v$), summing them ($+/ \bullet v$), and applying the exponential function ($*+/ \bullet v$). Thus:

$$\times/v \leftrightarrow *+/ \bullet v$$

Since the exponential function \circ is the inverse of the natural logarithm \bullet , the general form suggested by the right side of the identity is:

$$IG \ F/G \ V$$

where IG is the function inverse to G .

Using \wedge and \vee to denote the functions *and* and *or*, and \sim to denote the self-inverse function of logical negation, we may express DeMorgan's laws for an arbitrary number of elements by:

$$\begin{array}{l} \wedge/B \leftrightarrow \sim\vee/\sim B \\ \vee/B \leftrightarrow \sim\wedge/\sim B \end{array}$$

The elements of B are, of course, restricted to the boolean values 0 and 1 . Using the relation symbols to denote *functions* (for example, $x < r$ yields 1 if x is less than r and 0 otherwise) we can express further dualities, such as:

$$\begin{array}{l} \neq/B \leftrightarrow \sim=/\sim B \\ =/B \leftrightarrow \sim\neq/\sim B \end{array}$$

Finally, using \sqcup and \sqcap to denote the *maximum* and *minimum* functions, we can express dualities which involve arithmetic negation:

$$\begin{array}{l} \sqcup/V \leftrightarrow -\sqcap/-V \\ \sqcap/V \leftrightarrow -\sqcup/-V \end{array}$$

It may also be noted that scan ($F\backslash$) may replace reduction ($F/$) in any of the foregoing dualities.

1.3 Subordination of Detail

As Babbage remarked in the passage cited by Cajori, brevity facilitates reasoning. Brevity is achieved by subordinating detail, and we will here consider three important ways of doing this: the use of arrays, the assignment of names to functions and variables, and the use of operators.

We have already seen examples of the brevity

provided by one-dimensional arrays (vectors) in the treatment of duality, and further subordination is provided by matrices and other arrays of higher rank, since functions defined on vectors are extended systematically to arrays of higher rank.

In particular, one may specify the axis to which a function applies. For example, $\phi[1]_M$ acts along the first axis of a matrix M to reverse each of the columns, and $\phi[2]_M$ reverses each row; $M.[1]_M$ concatenates columns (placing M above M), and $M.[2]_M$ concatenates rows; and $+/[1]_M$ sums columns and $+/[2]_M$ sums rows. If no axis is specified, the function applies along the last axis. Thus $+/M$ sums rows. Finally, reduction and scan along the *first* axis may be denoted by the symbols \circ and $\circ\backslash$.

Two uses of names may be distinguished: *constant* names which have fixed referents are used for entities of very general utility, and *ad hoc* names are assigned (by means of the symbol \sim) to quantities of interest in a narrower context. For example, the constant (name) 144 has a fixed referent, but the names *CRATE*, *LAYER*, and *ROW* assigned by the expressions

$$\begin{array}{l} CRATE \sim 144 \\ LAYER \sim CRATE \sim 8 \\ ROW \sim LAYER \sim 3 \end{array}$$

are ad hoc, or *variable* names. Constant names for vectors are also provided, as in $2 \ 3 \ 5 \ 7 \ 11$ for a numeric vector of five elements, and in ' $ABCDE$ ' for a character vector of five elements.

Analogous distinctions are made in the names of functions. Constant names such as $+$, \times , and $*$ are assigned to so-called *primitive* functions of general utility. The detailed definitions, such as $+/M_B M$ for $M \times M$ and $\times/M_B M$ for $M \times M$, are subordinated by the constant names \times and $*$.

Less familiar examples of constant function names are provided by the comma which *catenates* its arguments as illustrated by:

$$(15),(\phi 5) \leftrightarrow 1 \ 2 \ 3 \ 4 \ 5 \ 5 \ 4 \ 3 \ 2 \ 1$$

and by the *base-representation* function τ , which produces a representation of its right argument in the radix specified by its left argument. For example:

$$\begin{array}{l} 2 \ 2 \ 2 \ \tau \ 3 \leftrightarrow 0 \ 1 \ 1 \\ 2 \ 2 \ 2 \ \tau \ 4 \leftrightarrow 1 \ 0 \ 0 \\ BN \# 2 \ 2 \ \tau \ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \\ BN \\ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \\ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ BN, \# BN \\ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \end{array}$$

The matrix BN is an important one, since it can be viewed in several ways. In addition to representing the binary numbers, the columns represent all subsets of a set of three elements, as well as the en-

tries in a truth table for three boolean arguments. The general expression for n elements is easily seen to be $(N \# 2) \tau(1:2:N)-1$, and we may wish to assign an ad hoc name to this function. Using the direct definition form (Appendix B), the name τ is assigned to this function as follows:

$\tau: (\omega \# 2) \tau(1:2:\omega)-1$

A.2

The symbol ω represents the argument of the function; in the case of two arguments the left is represented by ω . Following such a definition of the function τ , the expression T^3 yields the boolean matrix B_N shown above.

Three expressions, separated by colons, are also used to define a function as follows: the middle expression is executed first; if its value is zero the first expression is executed, if not, the last expression is executed. This form is convenient for recursive definitions, in which the function is used in its own definition. For example, a function which produces binomial coefficients of an order specified by its argument may be defined recursively as follows:

$BC: (X, 0) + (0, X+BC \omega-1); \omega=0:1$

A.3

Thus $BC\ 0 \leftrightarrow 1$ and $BC\ 1 \leftrightarrow 1\ 1$ and $BC\ n \leftrightarrow 1\ 4\ 6\ 4\ 1$.

The term *operator*, used in the strict sense defined in mathematics rather than loosely as a synonym for *function*, refers to an entity which applies to functions to produce functions; an example is the derivative operator.

We have already met two operators, *reduction*, and *scan*, denoted by $/$ and \backslash , and seen how they contribute to brevity by applying to different functions to produce families of related functions such as $+/$ and $\times/$ and $^{1/}$. We will now illustrate the notion further by introducing the *inner product* operator denoted by a period. A function (such as $+/$) produced by an operator will be called a *derived function*.

If P and Q are two vectors, then the inner product $+\cdot\cdot$ is defined by:

$$P+\cdot\cdot Q \leftrightarrow +/P\times Q$$

and analogous definitions hold for function pairs other than $+$ and \times . For example:

$$\begin{array}{l} P+2\ 3\ 5 \\ Q+2\ 1\ 2 \\ P+\cdot\cdot Q \end{array}$$

17

$$\begin{array}{l} P\times\cdot\cdot Q \\ 300 \\ P\cdot\cdot+Q \end{array}$$

Each of the foregoing expressions has at least one useful interpretation: $P+\cdot\cdot Q$ is the total cost of order quantities Q for items whose prices are given by P ; because P is a vector of primes, $P\times\cdot\cdot Q$ is the number whose prime decomposition is given by the exponents Q ; and if P gives distances from a source

to transhipment points and Q gives distances from the transhipment points to the destination, then $P\cdot\cdot+Q$ gives the minimum distance possible. The function $+\cdot\cdot$ is equivalent to the inner product or dot product of mathematics, and is extended to matrices as in mathematics. Other cases such as $\cdot\cdot\cdot$ are extended analogously. For example, if τ is the function defined by A.2, then:

$$\begin{array}{c} T^3 \\ \begin{array}{ccccccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array} \\ P+\cdot\cdot\cdot T^3 \\ \begin{array}{cccccc} 0 & 5 & 3 & 8 & 2 & 7 \\ 5 & 10 & 15 & 20 & 25 & 30 \end{array} \end{array} \quad \begin{array}{c} P\times\cdot\cdot\cdot T^3 \\ \begin{array}{cccccc} 1 & 5 & 3 & 15 & 2 & 10 \\ 5 & 15 & 25 & 45 & 10 & 30 \end{array} \end{array}$$

These examples bring out an important point: if S is boolean, then $P+\cdot\cdot\cdot S$ produces sums over subsets of P specified by 1 's in S , and $P\times\cdot\cdot\cdot S$ produces products over subsets.

The phrase $\cdot\cdot\cdot$ is a special use of the inner product operator to produce a derived function which yields products of each element of its left argument with each element of its right. For example:

$$\begin{array}{c} 2\ 3\ 5\ \cdot\cdot\cdot 15 \\ 2\ 4\ 6\ 8\ 10 \\ 3\ 6\ 9\ 12\ 15 \\ 5\ 10\ 15\ 20\ 25 \end{array}$$

The function $\cdot\cdot\cdot$ is called *outer product*, as it is in tensor analysis, and functions such as $\cdot\cdot\cdot\cdot$ and $\cdot\cdot\cdot\cdot\cdot$ and $\cdot\cdot\cdot\cdot\cdot\cdot$ are defined analogously, producing "function tables" for the particular functions. For example:

$$\begin{array}{c} D+0\ 1\ 2\ 3 \\ D\cdot\cdot\cdot ID \\ \begin{array}{cccc} 0 & 1 & 2 & 3 \\ 1 & 1 & 2 & 3 \\ 2 & 2 & 2 & 3 \\ 3 & 3 & 3 & 3 \end{array} \end{array} \quad \begin{array}{c} D+0\cdot2D \\ D\cdot\cdot\cdot 2D \\ \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{array} \end{array} \quad \begin{array}{c} D\cdot\cdot\cdot ID \\ \begin{array}{cccc} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{array} \end{array}$$

The symbol $:$ denotes the binomial coefficient function, and the table $D\cdot\cdot\cdot ID$ is seen to contain Pascal's triangle with its apex at the left; if extended to negative arguments (as with $D+0\cdot2D\cdot1\cdot0\cdot1\cdot2\cdot3$) it will be seen to contain the triangular and higher-order figurate numbers as well. This extension to negative arguments is interesting for other functions as well. For example, the table $D\cdot\cdot\cdot 2D$ consists of four quadrants separated by a row and a column of zeros, the quadrants showing clearly the rule of signs for multiplication.

Patterns in these function tables exhibit other properties of the functions, allowing brief statements of proofs by exhaustion. For example, commutativity appears as a symmetry about the diagonal. More precisely, if the result of the transpose function τ (which reverses the order of the axes of its argument) applied to a table $T+D\cdot\cdot\cdot ID$ agrees with T , then the function τ is commutative on the domain. For example, $T=\tau T+D\cdot\cdot\cdot ID$ produces a table of 1 's because τ is commutative.

Corresponding tests of associativity require rank 3 tables of the form $D \circ . I(D \circ , ID)$ and $(D \circ , ID) \circ . ID$. For example:

$D \circ . A(D \circ , ID)$	$(D \circ , AD) \circ . AD$	$D \circ . S(D \circ , SD)$	$(D \circ , SD) \circ . SD$
0 0	0 0	1 1	0 1
0 0	0 0	1 1	0 1
0 0	0 0	1 1	1 1
0 1	0 1	0 1	0 1

1.4 Economy

The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind. Economy of notation is therefore important.

Economy requires that a large number of ideas be expressible in terms of a relatively small vocabulary. A fundamental scheme for achieving this is the introduction of grammatical rules by which meaningful phrases and sentences can be constructed by combining elements of the vocabulary.

This scheme may be illustrated by the first example treated -- the relatively simple and widely useful notion of the sum of the first n integers was not introduced as a primitive, but as a phrase constructed from two more generally useful notions, the function ! for the production of a vector of integers, and the function +/ for the summation of the elements of a vector. Moreover, the derived function +/ is itself a phrase, summation being a derived function constructed from the more general notion of the reduction operator applied to a particular function.

Economy is also achieved by generality in the functions introduced. For example, the definition of the factorial function denoted by : is not restricted to integers, and the gamma function of x may therefore be written as $:x\text{-}1$. Similarly, the *relations* defined on all real arguments provide several important logical functions when applied to boolean arguments: exclusive-or (^), material implication (≤), and equivalence (=).

The economy achieved for the matters treated thus far can be assessed by recalling the vocabulary introduced:

```

/   o   φ   T   ,
\   \   .
+--×*+*!Γ|Ω
××~≤≤≥≥≥

```

The five functions and three operators listed in the first two rows are of primary interest, the remaining familiar functions having been introduced to illustrate the versatility of the operators.

A significant economy of symbols, as opposed to economy of functions, is attained by allowing any symbol to represent both a *monadic* function (i.e.

a function of one argument) and a *dyadic* function, in the same manner that the minus sign is commonly used for both subtraction and negation. Because the two functions represented may, as in the case of the minus sign, be related, the burden of remembering symbols is eased.

For example, $x \cdot y$ and $\cdot y$ represent power and exponential, $x \cdot y$ and $\cdot y$ represent base x logarithm and natural logarithm, $x \cdot y$ and $\cdot y$ represent division and reciprocal, and $x:y$ and $:y$ represent the binomial coefficient function and the factorial (that is, $x:y++(1:y)+(1:x)\times(1:y-x)$). The symbol \circ used for the dyadic function of replication also represents a monadic function which gives the shape of the argument (that is, $x++\text{o}x:y$), the symbol \diamond used for the monadic reversal function also represents the dyadic *rotate* function exemplified by $2\diamond 15++3 4 5 1 2$, and by $-2\diamond 15++4 5 1 2 3$, and finally, the comma represents not only catenation, but also the monadic *ravel*, which produces a vector of the elements of its argument in "row-major" order. For example:

$\frac{T}{2}$	$\frac{T}{2}$
0 0 1 1 1	0 0 1 1 0 1 0 1
0 1 0 1	

Simplicity of the grammatical rules of a notation is also important. Because the rules used thus far have been those familiar in mathematical notation, they have not been made explicit, but two simplifications in the order of execution should be remarked:

- (1) All functions are treated alike, and there are no rules of precedence such as \times being executed before $+$.
- (2) The rule that the right argument of a monadic function is the value of the entire expression to its right, implicit in the order of execution of an expression such as $\text{SIN LOG } 1:N$, is extended to dyadic functions.

The second rule has certain useful consequences in reduction and scan. Since $F:v$ is equivalent to placing the function F between the elements of v , the expression $-v:v$ gives the alternating sum of the elements of v , and $+v:v$ gives the alternating product. Moreover, if b is a boolean vector, then $\text{<}b$ "isolates" the first 1 in b , since all elements following it become 0. For example:

```
<\0 0 1 1 0 1 1 ++ 0 0 1 0 0 0 0
```

Syntactic rules are further simplified by adopting a single form for all dyadic functions, which appear between their arguments, and for all monadic functions, which appear before their arguments. This contrasts with the variety of rules in mathematics. For example, the symbols for the monadic functions of negation, factorial, and mag-

nitude precede, follow, and surround their arguments, respectively. Dyadic functions show even more variety.

1.5 Amenability to Formal Proofs

The importance of formal proofs and derivations is clear from their role in mathematics. Section 4 is largely devoted to formal proofs, and we will limit the discussion here to the introduction of the forms used.

Proof by exhaustion consists of exhaustively examining all of a finite number of special cases. Such exhaustion can often be simply expressed by applying some outer product to arguments which include all elements of the relevant domain. For example, if $D = \{0, 1\}$, then $D \times D$ gives all cases of application of the *and* function. Moreover, DeMorgan's law can be proved exhaustively by comparing each element of the matrix $D \times D$ with each element of $\sim(\sim D) \times \sim(\sim D)$ as follows:

	$D \times D$	$\sim(\sim D) \times \sim(\sim D)$
0 0	0 0	0 0
0 1	0 1	0 1
1 0	($D \times D$) = $\sim(\sim D) \times \sim(\sim D)$	($D \times D$) = $\sim(\sim D) \times \sim(\sim D)$
1 1	$\sim(\sim D \times D) = \sim(\sim(\sim D) \times \sim(\sim D))$	$\sim(\sim D \times D) = \sim(\sim(\sim D) \times \sim(\sim D))$
1		

Questions of associativity can be addressed similarly, the following expressions showing the associativity of *and* and the non-associativity of *not-and*:

1	$\wedge / ((D \times D) \times \sim D) = (D \times \wedge (D \times \sim D))$
0	$\wedge / ((D \times \sim D) \times \sim D) = (D \times \sim (D \times \sim D))$

A proof by a sequence of identities is presented by listing a sequence of expressions, annotating each expression with the supporting evidence for its equivalence with its predecessor. For example, a formal proof of the identity A.1 suggested by the first example treated would be presented as follows:

$\wedge / 1 N$	
$\wedge / \phi 1 N$	
$((\wedge / 1 N) + (\wedge / \phi 1 N)) * 2$	+ is associative and commutative $(X+X)*2 \leftrightarrow X$
$((\wedge / 1 N) + (\wedge / \phi 1 N)) * 2$	+ is associative and commutative Lemma
$((\wedge / ((N+1) \times N)) * 2$	Definition of \times
$((N+1) \times N) * 2$	

The fourth annotation above concerns an identity which, after observation of the pattern in the special case $(1 \times 1) + (\phi 1 \times 1)$, might be considered obvious or might be considered worthy of formal proof in a separate lemma.

Inductive proofs proceed in two steps: 1) some identity (called the *induction hypothesis*) is assumed true for a fixed integer value of some parameter n and this assumption is used to prove that the identity also holds for the value $n+1$, and 2) the identity is shown to hold for some integer value k . The conclusion is that the identity holds for all integer values of n which equal or exceed k .

Recursive definitions often provide convenient bases for inductive proofs. As an example we will use the recursive definition of the binomial coefficient function BC given by A.3 in an inductive proof showing that the sum of the binomial coefficients of order n is 2^n . As the induction hypothesis we assume the identity:

$$\wedge / BC \ N \leftrightarrow 2^N$$

and proceed as follows:

$\wedge / BC \ N+1$	
$\wedge / (X, 0) + (\wedge / X, BC \ N)$	+ is associative and commutative $0+Y \leftrightarrow Y$
$(\wedge / X, 0) + (\wedge / 0, X)$	$Y+Y \leftrightarrow 2 \times Y$
$2 \times X$	Definition of X
$2 \times \wedge / BC \ N$	Induction hypothesis
2×2^N	Property of Power ($*$)
2^{N+1}	

It remains to show that the induction hypothesis is true for some integer value of n . From the recursive definition A.3, the value of $BC \ 0$ is the value of the rightmost expression, namely 1. Consequently, $\wedge / BC \ 0$ is 1, and therefore equals 2^0 .

We will conclude with a proof that DeMorgan's law for scalar arguments, represented by:

$$A \wedge B \leftrightarrow \sim(\sim A) \vee(\sim B)$$

A.4

and proved by exhaustion, can indeed be extended to vectors of arbitrary length as indicated earlier by the putative identity:

$$\wedge / V \leftrightarrow \sim \sim / \sim V$$

A.5

As the induction hypothesis we will assume that A.5 is true for vectors of length $(\rho V)-1$.

We will first give formal recursive definitions of the derived functions *and-reduction* and *or-reduction* ($\wedge /$ and $\vee /$), using two new primitives, *indexing*, and *drop*. Indexing is denoted by an expression of the form $x[i]$, where i is a single index or array of indices of the vector x . For example, if $x = [3 \ 5 \ 7]$, then $x[2]$ is 5, and $x[2 \ 1]$ is 3 5. Drop is denoted by $x[k]$ and is defined to drop ρx (i.e., the magnitude of x) elements from x , from the head if $k > 0$ and from the tail if $k < 0$. For example, ρx is 5 7 and $\rho_{-2} x$ is 3 5. The *take* function (to be used later) is denoted by $+$ and is defined analogously. For example, $\rho_3 x$ is 2 3 5 and $\rho_{-3} x$ is 3 5 7.

The following functions provide formal definitions of *and-reduction* and *or-reduction*:

$$\begin{aligned} ANDRED : & w[1] \wedge ANDRED \ 1 \leftrightarrow 0 = \rho w : 1 \\ ORRED : & w[1] \vee ORRED \ 1 \leftrightarrow 0 = \rho w : 0 \end{aligned}$$

A.6

A.7

The inductive proof of A.5 proceeds as follows:

\wedge / V	
$(V[1]) \wedge (\wedge / 1 \vee V)$	
$\sim(\sim V[1]) \vee(\sim \sim / 1 \vee V)$	
$\sim(\sim V[1]) \vee(\sim \sim / \sim 1 \vee V)$	
$\sim(\sim V[1]) \vee(\sim / \sim 1 \vee V)$	$\sim \sim X \leftrightarrow X$
$\sim \sim / (\sim V[1]), (\sim 1 \vee V)$	A.7
$\sim \sim / (\sim V[1], 1 \vee V)$	\vee distributes over \sim
$\sim \sim / \sim V$	Definition of \sim (catenation)

2. Polynomials

If c is a vector of coefficients and x is a scalar, then the polynomial in x with coefficients c may be written simply as $+/c \cdot x^{-1+1\alpha c}$, or $+/(x^{-1+1\alpha c}) \cdot c$, or $(x^{-1+1\alpha c}) \cdot +. \cdot c$. However, to apply to a non-scalar array of arguments x , the power function $*$ should be replaced by the power table $\cdot \cdot *$ as shown in the following definition of the polynomial function:

$$P:=(w \cdot \cdot *^{-1+1\alpha a}) \cdot \cdot x^a \quad B.1$$

For example, $1 \ 3 \ 3 \ 1 \ P \ 0 \ 1 \ 2 \ 3 \ 4 \leftrightarrow 1 \ 8 \ 27 \ 64 \ 125$. If α is replaced by $1+\alpha$, then the function applies also to matrices and higher dimensional arrays of sets of coefficients representing (along the leading axis of a) collections of coefficients of different polynomials.

This definition shows clearly that the polynomial is a linear function of the coefficient vector. Moreover, if a and w are vectors of the same shape, then the pre-multiplier $w \cdot \cdot *^{-1+1\alpha a}$ is the Vandermonde matrix of w and is therefore invertible if the elements of w are distinct. Hence if c and x are vectors of the same shape, and if $r+c \in X$, then the inverse (curve-fitting) problem is clearly solved by applying the matrix inverse function $\#$ to the Vandermonde matrix and using the identity:

$$C \leftrightarrow (\#X \cdot \cdot *^{-1+1\alpha X}) \cdot \cdot Y$$

2.1 Products of Polynomials

The "product of two polynomials b and c " is commonly taken to mean the coefficient vector d such that:

$$D \ P \ X \leftrightarrow (B \ P \ X) \times (C \ P \ X)$$

It is well-known that d can be computed by taking products over all pairs of elements from b and c and summing over subsets of these products associated with the same exponent in the result. These products occur in the function table $B \cdot \cdot \cdot \cdot C$, and it is easy to show informally that the powers of x associated with the elements of $B \cdot \cdot \cdot \cdot C$ are given by the addition table $E = (-1+1\alpha B) \cdot \cdot \cdot \cdot (-1+1\alpha C)$. For example:

X^{+2}		
$B+3$	1	2
$C+2$	0	3
$B+(-1+1\alpha B) \cdot \cdot \cdot \cdot (-1+1\alpha C)$		
$B \cdot \cdot \cdot \cdot C$	E	$X \cdot \cdot \cdot \cdot$
6 0 9	0 1 2	1 2 4
2 0 3	1 2 3	2 4 8
4 0 6	2 3 4	4 8 16
6 0 9	3 4 5	8 16 32
$+/, (B \cdot \cdot \cdot \cdot C) \times X \cdot \cdot \cdot \cdot$		
518	$(B \ P \ X) \times (C \ P \ X)$	
518		

The foregoing suggests the following identity, which will be established formally in Section 4:

$$(B \ P \ X) \times (C \ P \ X) \leftrightarrow +/, (B \cdot \cdot \cdot \cdot C) \times X \cdot \cdot \cdot \cdot (-1+1\alpha B) \cdot \cdot \cdot \cdot (-1+1\alpha C) \quad B.2$$

Moreover, the pattern of the exponent table E shows that elements of $B \cdot \cdot \cdot \cdot C$ lying on diagonals are associated with the same power, and that the coefficient vector of the product polynomial is therefore given by sums over these diagonals. The table $B \cdot \cdot \cdot \cdot C$ therefore provides an excellent organization for the manual computation of products of polynomials. In the present example these sums give the vector $D = 6 \ 2 \ 13 \ 9 \ 6 \ 9$, and $D \ P \ X$ may be seen to equal $(B \ P \ X) \times (C \ P \ X)$.

Sums over the required diagonals of $B \cdot \cdot \cdot \cdot C$ can also be obtained by bordering it by zeros, skewing the result by rotating successive rows by successive integers, and then summing the columns. We thus obtain a definition for the polynomial product function as follows:

$$PP: +/(1-1\alpha a) \# a \cdot \cdot \cdot \cdot w, 1+0\alpha a$$

We will now develop an alternative method based upon the simple observation that if $B \ P \ P \ C$ produces the product of polynomials b and c , then $P \ P$ is linear in both of its arguments. Consequently,

$$PP: a \cdot \cdot \cdot \cdot A \cdot \cdot \cdot \cdot w$$

where A is an array to be determined. A must be of rank 3, and must depend on the exponents of the left argument $(-1+1\alpha a)$, of the result $(-1+1\alpha b, w)$, and of the right argument. The "deficiencies" of the right exponent are given by the difference table $(1\alpha 1+a, w) \cdot \cdot \cdot \cdot -1\alpha w$, and comparison of these values with the left exponents yields A . Thus

$$A+(-1+1\alpha a) \cdot \cdot \cdot \cdot = ((1\alpha 1+a, w) \cdot \cdot \cdot \cdot -1\alpha w)$$

and

$$PP: a \cdot \cdot \cdot \cdot x((-1+1\alpha a) \cdot \cdot \cdot \cdot = ((1\alpha 1+a, w) \cdot \cdot \cdot \cdot -1\alpha w) \cdot \cdot \cdot \cdot w$$

Since $a \cdot \cdot \cdot \cdot A$ is a matrix, this formulation suggests that if $D = B \ P \ P \ C$, then c might be obtained from D by pre-multiplying it by the inverse matrix $(BB \cdot \cdot \cdot \cdot A)$, thus providing division of polynomials. Since $B \cdot \cdot \cdot \cdot A$ is not square (having more rows than columns), this will not work, but by replacing $N+B \cdot \cdot \cdot \cdot A$ by either its leading square part $(2\rho L/\rho M)+M$, or by its trailing square part $(-2\rho L/\rho M)+N$, one obtains two results, one corresponding to division with low-order remainder terms, and the other to division with high-order remainder terms.

2.2 Derivative of a Polynomial

Since the derivative of x^N is $N \times x^{N-1}$, we may use the rules for the derivative of a sum of functions and of a product of a function with a constant, to show that the derivative of the polynomial $c \ P \ X$ is the polynomial $(1+cx^{-1+1\alpha c}) \ P \ X$. Using this result it is clear that the integral is the polynomial $(A, C+1\alpha C) \ P \ X$, where A is an arbitrary scalar constant. The expression $1+cx^{-1+1\alpha c}$ also yields the

coefficients of the derivative, but as a vector of the same shape as c and having a final zero element.

2.3 Derivative of a Polynomial with Respect to Its Roots

If R is a vector of three elements, then the derivatives of the polynomial $x^3 - R$ with respect to each of its three roots are $-(x-R[2]) \times (x-R[3])$, and $-(x-R[1]) \times (x-R[3])$, and $-(x-R[1]) \times (x-R[2])$. More generally, the derivative of $x^k - R$ with respect to $R[J]$ is simply $-(x-R) \times \dots \times I_{\neq J} \times R$, and the vector of derivatives with respect to each of the roots is $-(x-R) \times \dots \times I_{\neq 1} \times I_{\neq 1 \neq R}$.

The expression $x^k - R$ for a polynomial with roots R applies only to a scalar x , the more general expression being $x^k - R$. Consequently, the general expression for the matrix of derivatives (of the polynomial evaluated at $x[i]$ with respect to root $R[J]$) is given by:

$$-(X^k - R) \times \dots \times I_{\neq 1} \times I_{\neq 1 \neq R} \quad B.3$$

2.4 Expansion of a Polynomial

Binomial expansion concerns the development of an identity in the form of a polynomial in x for the expression $(X+Y)^n$. For the special case of $Y=1$ we have the well-known expression in terms of the binomial coefficients of order n :

$$(X+1)^n \leftrightarrow ((0, \dots, n) \otimes X)$$

By extension we speak of the expansion of a polynomial as a matter of determining coefficients d such that:

$$C \otimes X + Y \leftrightarrow D \otimes X$$

The coefficients d are, in general, functions of y . If $y=1$ they again depend only on binomial coefficients, but in this case on the several binomial coefficients of various orders, specifically on the matrix $J_{0..1} \otimes J_{-1..1} \otimes C$.

For example, if $C = [1 \ 2 \ 3]$, and $C \otimes X + Y \leftrightarrow D \otimes X$, then D depends on the matrix:

$$\begin{matrix} 0 & 1 & 2 & 3 & \dots & 0 & 1 & 2 & 3 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{matrix}$$

and D must clearly be a weighted sum of the columns, the weights being the elements of C . Thus:

$$D = (J_{0..1} \otimes J_{-1..1} \otimes C) \times C$$

Jotting down the matrix of coefficients and performing the indicated matrix product provides a quick and reliable way to organize the otherwise messy manual calculation of expansions.

If B is the appropriate matrix of binomial coefficients, then $D = B \times C$, and the expansion function is clearly linear in the coefficients c . Moreover, expansion for $y=-1$ must be given by the inverse matrix \bar{B} , which will be seen to contain the alternating binomial coefficients. Finally, since:

$$C \otimes X^{k+1} \leftrightarrow C \otimes (X+k) + 1 \leftrightarrow (B \times C) \otimes (X+k)$$

it follows that the expansion for positive integer values of y must be given by products of the form:

$$B \times B \times \dots \times B \times C$$

where the B occurs y times.

Because \times is associative, the foregoing can be written as $M \times C$, where M is the product of y occurrences of B . It is interesting to examine the successive powers of B , computed either manually or by machine execution of the following inner product power function:

$$IPP: a \times \dots \times a \quad IPP \omega-1 : \omega=0 : j \dots -j+1 \times 1 + p B$$

Comparison of $B \otimes K$ with B for a few values of K shows an obvious pattern which may be expressed as:

$$B \otimes K \leftrightarrow B \times K \otimes -J \dots -J+1 \times 1 + p B$$

The interesting thing is that the right side of this identity is meaningful for non-integer values of K , and, in fact, provides the desired expression for the general expansion $C \otimes X + Y$:

$$C \otimes (X+Y) \leftrightarrow (((J \dots 1) \times Y) \otimes -J \dots -J+1 + p C) \times C \otimes X \quad B.4$$

The right side of B.4 is of the form $(M \times C) \otimes X$, where M itself is of the form $B \times Y \times E$ and can be displayed informally (for the case $Y=C$) as follows:

$$\begin{matrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{matrix} \times Y \times \begin{matrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$

Since $Y \otimes K$ multiplies the single-diagonal matrix $B \times (K \otimes E)$, the expression for M can also be written as the inner product $(Y \otimes J) \times \dots \times T$, where T is a rank 3 array whose k th plane is the matrix $B \times (K \otimes E)$. Such a rank three array can be formed from an upper triangular matrix M by making a rank 3 array whose first plane is M (that is, $(1 \dots (1+pM) \dots M)$) and rotating it along the first axis by the matrix $J_{0..1}$, whose k th superdiagonal has the value $-K$. Thus:

$$DS: (I \dots -I) \Phi[1] (1 \dots 1 + p \omega) \dots \times \omega \quad B.5$$

$$DS \ K \otimes -1 \dots 1$$

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{matrix}$$

$$\begin{matrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

Substituting these results in B.4 and using the associativity of \times , we have the following identity for the expansion of a polynomial, valid for non-integer as well as integer values of y :

$$C \otimes X + Y \leftrightarrow ((Y \otimes J) \times \dots \times (DS \ J \otimes -1 \dots 1 + p C) \times C) \otimes X \quad B.6$$

For example:

```

Y=3
C=3 1 4 2
M=(Y+J)+,xDS J+=1J+=-1+1pC
M
1 3 9 27
0 1 6 27
0 0 1 9
0 0 0 1
M+.xC
96 79 22 2
(M+.xC) P X+2
358 C P X+Y
358

```

3. Representations

The subjects of mathematical analysis and computation can be *represented* in a variety of ways, and each representation may possess particular advantages. For example, a positive integer n may be represented simply by n check-marks; less simply, but more compactly, in Roman numerals; even less simply, but more conveniently for the performance of addition and multiplication, in the decimal system; and less familiarly, but more conveniently for the computation of the least common multiple and the greatest common divisor, in the prime decomposition scheme to be discussed here.

Graphs, which concern connections among a collection of elements, are an example of a more complex entity which possesses several useful representations. For example, a simple directed graph of n elements (usually called *nodes*) may be represented by an n by n boolean matrix B (usually called an *adjacency matrix*) such that $B(I,J)=1$ if there is a connection from node I to node J . Each connection represented by a 1 in B is called an *edge*, and the graph can also be represented by a $+/_B$ by n matrix in which each row shows the nodes connected by a particular edge.

Functions also admit different useful representations. For example, a permutation function, which yields a reordering of the elements of its vector argument x , may be represented by a *permutation vector* P such that the permutation function is simply $x[P]$, by a *cycle* representation which presents the structure of the function more directly, by the boolean matrix $B+P=1_{pP}$ such that the permutation function is $B+.x$, or by a *radix* representation R which employs one of the columns of the matrix $1+(\Phi(N))^{-1}1_{N+pX}$, and has the property that $21+/R-1$ is the parity of the permutation represented.

In order to use different representations conveniently, it is important to be able to express the transformations between representations clearly and precisely. Conventional mathematical notation is often deficient in this respect, and the present section is devoted to developing expressions for the transformations between representations useful in a variety of topics: number systems, polynomials, permutations, graphs, and boolean algebra.

3.1 Number Systems

We will begin the discussion of representations with a familiar example, the use of different representations of positive integers and the transformations between them. Instead of the *positional* or *base-value* representations commonly treated, we will use *prime decomposition*, a representation whose interesting properties make it useful in introducing the idea of logarithms as well as that of number representation [6, Ch.16].

If P is a vector of the first p_P primes and E is a vector of non-negative integers, then E can be used to represent the number $P \times .^* E$, and all of the integers $1 \leq P$ can be so represented. For example, $2 3 5 7 \times .^* 0 0 0 0$ is 1 and $2 3 5 7 \times .^* 1 1 0 0$ is 6 and:

```

P
2 3 5 7
ME
0 1 0 2 0 1 0 3 0 1
0 0 1 0 0 1 0 0 2 0
0 0 0 0 1 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0
Px.^*ME
1 2 3 4 5 6 7 8 9 10

```

The similarity to logarithms can be seen in the identity:

$$*/Px.^*ME \leftrightarrow Px.^*/ME$$

which may be used to effect multiplication by addition.

Moreover, if we define *gcd* and *lcm* to give the greatest common divisor and least common multiple of elements of vector arguments, then:

$GCD\ Px.^*ME \leftrightarrow Px.^*L/ME$ $LCM\ Px.^*ME \leftrightarrow Px.^*F/ME$	ME $2 1 0$ $3 1 2$ $2 2 0$ $1 2 3$	$V+Px.^*ME$ V $18900\ 7350\ 3087$ $GCD\ V$ 21 $Px.^*L/ME$ 21	$LCM\ V$ 926100 $Px.^*F/ME$ 926100
--	--	--	---

In defining the function *gcd*, we will use the operator */* with a boolean argument B (as in $B/$). It produces the *compression* function which selects elements from its right argument according to the ones in B . For example, $1 0 1 0 1/15$ is $1 3 5$. Moreover, the function $B/$ applied to a matrix argument compresses rows (thus selecting certain columns), and the function B_F compresses columns to select rows. Thus:

$$GCD: GCD\ M, (M+L/R)IR: 1 \geq pR+(w=0)/w: +/R$$

$$LCM: (\times/X) \div GCD\ X+(1+w), LCM\ 1+w: 0=pw: 1$$

The transformation to the value of a number from its prime decomposition representation (*VPR*) and the inverse transformation to the representation from the value (*RPV*) are given by:

$$VPR: \alpha x.^*w$$

$$RPV: D+\alpha RPV w+\alpha x.^*D: \wedge/\sim D+0=\alpha w:D$$

For example:

P VPR 2 1 3 1
 10500 P RRFV 10500
 2 1 3 1

3.2 Polynomials

Section 2 introduced two representations of a polynomial on a scalar argument x , the first in terms of a vector of coefficients c (that is, $+/c \times x^{-1+1\omega c}$), and the second in terms of its roots R (that is, \times/x^{-R}). The coefficient representation is convenient for adding polynomials ($c+d$) and for obtaining derivatives ($1+c \times x^{-1+1\omega c}$). The root representation is convenient for other purposes, including multiplication which is given by $R_1 \cdot R_2$.

We will now develop a function CPR (Coefficients from Roots) which transforms a roots representation to an equivalent coefficient representation, and an inverse function RFC . The development will be informal; a formal derivation of CPR appears in Section 4.

The expression for CPR will be based on Newton's symmetric functions, which yield the coefficients as sums over certain of the products over all subsets of the arithmetic negation (that is, $-R$) of the roots R . For example, the coefficient of the constant term is given by $\times/-R$, the product over the entire set, and the coefficient of the next term is a sum of the products over the elements of $-R$ taken $(\omega R)-1$ at a time.

The function defined by A.2 can be used to give the products over all subsets as follows:

$P + (-R) \times . * M + T \omega R$

The elements of P summed to produce a given coefficient depend upon the number of elements of R excluded from the particular product, that is, upon $+/\sim N$, the sum of the columns of the complement of the boolean "subset" matrix ωR .

The summation over P may therefore be expressed as $((0, \omega R) \circ . = +/\sim N) + . \times P$, and the complete expression for the coefficients c becomes:

$C + ((0, \omega R) \circ . = +/\sim N) + . \times (-R) \times . * M + T \omega R$

For example, if $R=2 3 5$, then

M $\begin{array}{ccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ (-R) \times . * M \\ 1 & -5 & -3 & 15 & -2 & 10 & 6 \\ -30 & 31 & -10 & 1 \end{array}$	$+/\sim N$ $\begin{array}{ccccccc} 3 & 2 & 2 & 1 & 2 & 1 & 1 & 0 \\ (0, \omega R) \circ . = +/\sim N \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$
--	--

The function CPR which produces the coefficients from the roots may therefore be defined and used as follows:

$CPR: ((0, \omega R) \circ . = +/\sim N) + . \times (-R) \times . * M + T \omega R$ C.1

$CPR 2 3 5$ $-30 31 -10 1$ $(CPR 2 3 5) P X+1 2 3 4 5 6 7 8$ $-8 0 0 -2 0 12 40 90$ $\times/X \circ . -2 3 5$ $-8 0 0 -2 0 12 40 90$

The inverse transformation RFC is more difficult, but can be expressed as a successive approximation scheme as follows:

```

RPC: (-1+1\omega 1+\omega)G \omega
G: (a-2)G \omega:TOL2/|Z+a STEP \omega:a-Z
STEP:(\omega(a\circ.-a)\times.*I\circ.-z1+1\omega a)+.\times(\omega\circ.*-1+1\omega \omega)+.\times\omega
D+C+CPR 2 3 5 7
210 -247 101 -17 1
TOL+1E-8
RFC C
7 5 2 3

```

The order of the roots in the result is, of course, immaterial. The final element of any argument of RFC must be 1, since any polynomial equivalent to $\times/X-R$ must necessarily have a coefficient of 1 for the high order term.

The foregoing definition of RFC applies only to coefficients of polynomials whose roots are all real. The left argument of G in RFC provides (usually satisfactory) initial approximations to the roots, but in the general case some at least must be complex. The following example, using the roots of unity as the initial approximation, was executed on an APL system which handles complex numbers:

```

(+0\omega J2\times(-1+1\omega N)+N+\omega 1+\omega)G\omega
D+C+CPR 1J1 1J-1 1J2 1J-2
10 -14 11 -4 1
RFC C
1J-1 1J2 1J1 1J-2

```

The monadic function \circ used above multiplies its argument by pi.

In Newton's method for the root of a scalar function F , the next approximation is given by $A+A-(F A)+DF A$, where DF is the derivative of F . The function $STEP$ is the generalization of Newton's method to the case where F is a vector function of a vector. It is of the form $(\omega M)+.\times B$, where B is the value of the polynomial with coefficients ω , the original argument of RFC , evaluated at a , the current approximation to the roots; analysis similar to that used to derive B.3 shows that M is the matrix of derivatives of a polynomial with roots a , the derivatives being evaluated at a .

Examination of the expression for M shows that its off-diagonal elements are all zero, and the expression $(\omega M)+.\times B$ may therefore be replaced by $B+D$, where D is the vector of diagonal elements of M . Since $(I,J)+N$ drops I rows and J columns from a matrix N , the vector D may be expressed as $\times/0 1+(-1+1\omega a)\Phi a\circ.-a$; the definition of the function $STEP$ may therefore be replaced by the more efficient definition:

$STEP: ((a\circ.*-1+1\omega a)+.\times\omega) + \times/0 1+(-1+1\omega a)\Phi a\circ.-a$ C.3

This last is the elegant method of Kerner [7]. Using starting values given by the left argument of G in C.2, it converges in seven steps (with a tolerance $TOL+1E-8$) for the sixth-order example given by Kerner.

3.3 Permutations

A vector p whose elements are some permutation of its indices (that is, $\wedge_{i=1}^n p_i = \wedge_{i=1}^n i$) will be called a *permutation* vector. If p is a permutation vector such that $(p \cdot x) = x \cdot p$, then $x \cdot p$ is a permutation of x , and p will be said to be the *direct representation* of this permutation.

The permutation $x \cdot p$ may also be expressed as $B \cdot x$, where B is the boolean matrix $B_{ij} = \delta_{ip_j}$. The matrix B will be called the *boolean representation* of the permutation. The transformations between direct and boolean representations are:

$$BFD: w \cdot \wedge_{i=1}^n p_i$$

$$DFB: w \cdot \wedge_{i=1}^n p_i$$

Because permutation is associative, the composition of permutations satisfies the following relations:

$$(X[D1][D2] \leftrightarrow X[(D1 \cdot D2)]) \\ B2 \cdot x(B1 \cdot x) \leftrightarrow (B2 \cdot xB1) \cdot x$$

The inverse of a boolean representation B is $\wedge B$, and the inverse of a direct representation is either $\wedge p$ or $\wedge_{i=1}^n p_i$. (The *grade* function \wedge grades its argument, giving a vector of indices to its elements in ascending order, maintaining existing order among equal elements. Thus $\wedge 3 \cdot 7 \cdot 1 \cdot 4$ is $3 \cdot 1 \cdot 4 \cdot 2$ and $\wedge 3 \cdot 7 \cdot 3 \cdot 4$ is $1 \cdot 3 \cdot 4 \cdot 2$. The *index-of* function \wedge determines the smallest index in its left argument of each element of its right argument. For example, ' $ABCDE$ ' \wedge ' $BABE$ ' is $2 \cdot 1 \cdot 2 \cdot 5$, and ' $BABE$ ' \wedge ' $ABCDE$ ' is $2 \cdot 1 \cdot 5 \cdot 5 \cdot 4$.)

The *cycle* representation also employs a permutation vector. Consider a permutation vector c and the segments of c marked off by the vector $c = \wedge C$. For example, if $c = 7 \cdot 3 \cdot 6 \cdot 5 \cdot 2 \cdot 1 \cdot 4$, then $c = \wedge C$ is $1 \cdot 1 \cdot 0 \cdot 0 \cdot 1 \cdot 1 \cdot 0$, and the blocks are:

```

7
3 6 5
2
1 4

```

Each block determines a "cycle" in the associated permutation in the sense that if R is the result of permuting x , then:

$R[7]$ is $X[7]$	$R[6]$ is $X[5]$	$R[5]$ is $X[3]$
$R[3]$ is $X[6]$	$R[4]$ is $X[1]$	$R[2]$ is $X[2]$
$R[1]$ is $X[4]$		$R[5]$ is $X[3]$

If the leading element of c is the smallest (that is, 1), then c consists of a single cycle, and the permutation of a vector x which it represents is given by $x[c] \rightarrow x[\wedge c]$. For example:

```

x + "ABCDEFG"
C + 1 7 6 5 2 4 3
x[c] -> x[w]
x
GDACBEGF

```

Since $x[c] \rightarrow x$ is equivalent to $x \rightarrow x[c]$, it follows that $x[c] -> x[\wedge c]$ is equivalent to $x + x[(\wedge c)[\wedge c]]$, and the direct representation vector p equivalent to c is therefore given (for the special case of a single cycle) by $p = (\wedge c)[\wedge c]$.

In the more general case, the rotation of the complete vector (that is, $\wedge c$) must be replaced by rotations of the individual subcycles marked off by $c = \wedge C$, as shown in the following definition of the transformation to direct from cycle representation:

$$DFC: (w[\wedge X + \wedge x = \wedge c])[\wedge w]$$

If one wishes to catenate a collection of disjoint cycles to form a single vector c such that $c = \wedge C$ marks off the individual cycles, then each cycle C_i must first be brought to *standard form* by the rotation $(\wedge_{i+1}^{i+L/C_i} \cdot L/C_i) \cdot C_i$, and the resulting vectors must be catenated in descending order on their leading elements.

The inverse transformation from direct to cycle representation is more complex, but can be approached by first producing the matrix of all powers of p up to the n th, that is, the matrix whose successive columns are p and $p \cdot p$ and $(p \cdot p \cdot p)$, etc. This is obtained by applying the function POW to the one-column matrix $p \cdot \wedge_{i=0}^n$ formed from p , where POW is defined and used as follows:

$$POW: POW D, (\wedge_{i=0}^n)[w]: s / \wedge w: w$$

```

D + D + DFC C + 7, 3 6 5, 2, 1 4
4 2 6 1 3 5 7
POW D + , 0
4 1 4 1 4 1 4
2 2 2 2 2 2 2
6 5 3 6 5 3 6
1 4 1 4 1 4 1
3 6 5 3 6 5 3
5 3 6 5 3 6 5
7 7 7 7 7 7 7

```

If $M = POW D \cdot \wedge_{i=0}^n$, then the cycle representation of D may be obtained by selecting from M only "standard" rows which begin with their smallest elements (SSR), by arranging these remaining rows in descending order on their leading elements (DOL), and then catenating the cycles in these rows (CIR). Thus:

$$CFD: CIR DOL SSR POW w \cdot \wedge_{i=0}^n$$

```

SSR: (\wedge M + 1 \wedge M - 1 \wedge w) / \wedge w
DOL: w[\wedge w]
CIR: (, 1, \wedge 0 1 + w = \wedge w) / , w

```

```

DFC C + 7, 3 6 5, 2, 1 4
4 2 6 1 3 5 7
CFD DFC C
7 3 6 5 2 1 4

```

In the definition of DOL , indexing is applied to matrices. The indices for successive coordinates are separated by semicolons, and a blank entry for any axis indicates that all elements along it are selected. Thus $w[1:1]$ selects column 1 of M .

The cycle representation is convenient for determining the number of cycles in the permutation represented ($(NC: + / w = \wedge c)$), the cycle lengths ($CL: x - 0, \wedge 1 + x + (\wedge w = \wedge c) / \wedge w$), and the *power* of the permutation ($PP: LCM CL w$). On the other hand, it is awkward for composition and inversion.

The $:N$ column vectors of the matrix $(\Phi : N)^T \wedge 1 + 1:N$ are all distinct, and therefore provide

a potential radix representation [8] for the $n!$ permutations of order n . We will use instead a related form obtained by increasing each element by 1:

$RR := 1 + (\phi \omega) \tau^{-1} + \dots \omega$

RR^4

1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4
1	1	2	2	3	3	1	1	2	2	3	3	1	1	2	2	3	3	1	1	2	2	3
1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Transformations between this representation and the direct form are given by:

$DFR := \omega[1], X + \omega[1] \leq X + DFR \quad 1 + \omega[0] = \rho \omega : \omega$
 $RFD := \omega[1], RFD \cdot X - \omega[1] \leq X + 1 + \omega[0] = \rho \omega : \omega$

Some of the characteristics of this alternate representation are perhaps best displayed by modifying DFR to apply to all columns of a matrix argument, and applying the modified function MF to the result of the function RR :

$MF := \omega[1], [1]X + \omega[[1 \rho X] \rho 1;] \leq X + MF \cdot 1 \quad 0 + \omega[0] = 1 + \rho \omega : \omega$
 $MF \cdot RR^4$

1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	
2	2	3	3	4	4	1	1	3	3	4	4	1	1	2	2	4	4	1	1	2	2	3
3	4	2	4	2	3	3	4	1	4	1	3	2	4	1	4	1	2	2	3	1	3	
4	3	4	2	3	2	4	3	4	1	3	1	4	2	4	1	2	1	3	2	3	1	

The direct permutations in the columns of this result occur in *lexical order* (that is, in ascending order on the first element in which two vectors differ); this is true in general, and the alternate representation therefore provides a convenient way for producing direct representations in lexical order.

The alternate representation also has the useful property that the parity of the direct permutation ρ is given by $2^{1+/-1+RFD} \rho$, where NIN represents the residue of n modulo N . The parity of a direct representation can also be determined by the function:

$PAR := 2^{1+/-1+/-1+/\rho \omega} \wedge \omega \wedge \omega \wedge \omega$

3.4 Directed Graphs

A simple directed graph is defined by a set of x nodes and a set of directed connections from one to another of pairs of the nodes. The directed connections may be conveniently represented by a κ by κ boolean *connection matrix* c in which $c[i;j]=1$ denotes a connection from the i th node to the j th.

For example, if the four nodes of a graph are represented by $N+'QRST'$, and if there are connections from node s to node e , from r to t , and from t to e , then the corresponding connection matrix is given by:

0	0	0	0
0	0	0	1
1	0	0	0
1	0	0	0

A connection from a node to itself (called a self-loop) is not permitted, and the diagonal of a connection matrix must therefore be zero.

If P is any permutation vector of order ρ_N , then

$N+NP$ is a reordering of the nodes, and the corresponding connection matrix is given by $c[P;P]$. We may (and will) without loss of generality use the numeric labels ρ_P for the nodes, because if n is any arbitrary vector of names for the nodes and ℓ is any list of numeric labels, then the expression $Q+N[\ell]$ gives the corresponding list of names and, conversely, $N[\ell]$ gives the list ℓ of numeric labels.

The connection matrix c is convenient for expressing many useful functions on a graph. For example, $+/c$ gives the *out-degrees* of the nodes, $+/_c$ gives the *in-degrees*, $+/_c$ gives the number of connections or *edges*, $*c$ gives a related graph with the directions of edges reversed, and cvc gives a related "symmetric" or "undirected" graph. Moreover, if we use the boolean vector $B+\vee/(11\rho c)\dots=L$ to represent the list of nodes ℓ , then $BV.c$ gives the boolean vector which represents the set of nodes directly reachable from the set B . Consequently, $cV.AC$ gives the connections for paths of length two in the graph c , and $cVcV.AC$ gives connections for paths of length one or two. This leads to the following function for the *transitive closure* of a graph, which gives all connections through paths of any length:

$TC := TC \cdot Z := \wedge_{i,j} \omega = Z + \omega \vee \omega V \wedge \omega \cdot Z$

Node j is said to be *reachable* from node i if $(TC c)[i;j]=1$. A graph is *strongly-connected* if every node is reachable from every node, that is \wedge_i,TC .

If $D+TC c$ and $D[i,i]=1$ for some i , then node i is reachable from itself through a path of some length; the path is called a *circuit*, and node i is said to be contained in a circuit.

A graph τ is called a *tree* if it has no circuits and its in-degrees do not exceed 1, that is, $\wedge_{i>1} \tau[i]$. Any node of a tree with an in-degree of 0 is called a *root*, and if $K++/0=+/\tau$, then τ is called a K -rooted tree. Since a tree is circuit-free, K must be at least 1. Unless otherwise stated, it is normally assumed that a tree is *singly-rooted* (that is, $K=1$); multiply-rooted trees are sometimes called *forests*.

A graph c covers a graph d if $\wedge_i, c \geq d$. If c is a strongly-connected graph and τ is a (singly-rooted) tree, then τ is said to be a *spanning tree* of c if c covers τ and if all nodes are reachable from the root of τ , that is,

$(\wedge_i, G \geq \tau) \wedge \wedge_i / R \vee R_V \wedge \wedge_i C \cdot \tau$

where R is the (boolean representation of the) root of τ .

A *depth-first spanning tree* [9] of a graph c is a spanning tree produced by proceeding from the root through immediate descendants in c , always choosing as the next node a descendant of the latest in the list of nodes visited which still possesses a descendant not in the list. This is a relatively

complex process which can be used to illustrate the utility of the connection matrix representation:

```
DPST:((1)~,=K) R w^K~,v~K+a=11+pw          C.4
R:(C,[1]a)Rw^P~,v~C+<\UAPV~,~a
:~v/P+(<\av.,~wv.,~U+~vfa)v.,~a
:a
```

Using as an example the graph *c* from [9]:

<i>G</i>	<i>1 DPST G</i>
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0	0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0	0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

The function *DPST* establishes the left argument of the recursion *R* as the one-row matrix representing the root specified by the left argument of *DPST*, and the right argument as the original graph with the connections *into* the root *x* deleted. The first line of the recursion *R* shows that it continues by appending on the top of the list of nodes thus far assembled in the left argument the next child *c*, and by deleting from the right argument all connections into the chosen child *c* except the one from its parent *p*. The child *c* is chosen from among those reachable from the chosen parent (*PV*, ~*w*), but is limited to those as yet untouched (*UAPV*, ~*w*), and is taken, arbitrarily, as the first of these (<\UAPV, ~*w*).

The determinations of *p* and *v* are shown in the second line, *p* being chosen from among those nodes which have children among the untouched nodes (~*v*, ~*w*). These are permuted to the order of the nodes in the left argument (~*v*, ~*wv*, ~*w*), bringing them into an order so that the last visited appears first, and *p* is finally chosen as the first of these.

The last line of *R* shows the final result to be the resulting right argument *w*, that is, the original graph with all connections into each node broken except for its parent in the spanning tree. Since the final value of *a* is a square matrix giving the nodes of the tree in reverse order as visited, substitution of *w*,~*a* (or, equivalently, *w*,~*a*) for *w* would yield a result of shape $1 \times n \times n$ containing the spanning tree followed by its "preordering" information.

Another representation of directed graphs often used, at least implicitly, is the list of all node pairs *v,w* such that there is a connection from *v* to *w*. The transformation to this list form from the connection matrix may be defined and used as follows:

```
LPC:(1,w)/1+Dx^-1+1x/D=pw
LPC C
0 0 1 1      1 1 2 3 3 4
0 1 0        3 4 3 2 4 1
```

However, this representation is deficient since it does not alone determine the number of nodes in the graph, although in the present example this is given by *LFC C* because the highest numbered node happens to have a connection. A related boolean representation is provided by the expression (*LPC C*) $\cdot\cdot\cdot=11+pC$, the first plane showing the out- and the second showing the in-connections.

An *incidence* matrix representation often used in the treatment of electric circuits [10] is given by the difference of these planes as follows:

```
IFC:-/(LFC w)\cdot\cdot\cdot=11+pw
```

For example:

<i>(LPC C)</i> $\cdot\cdot\cdot=11+pC$	<i>IFC C</i>
1 0 0 0	1 0 -1 0
1 0 0 0	1 0 0 -1
0 1 0 0	0 1 -1 0
0 0 1 0	0 -1 1 0
0 0 1 0	0 0 1 -1
0 0 0 1	-1 0 0 1
0 0 1 0	0 0 1 0
0 0 1 0	0 0 1 0
0 1 0 0	0 1 0 0
0 0 0 1	0 0 0 1
1 0 0 0	1 0 0 0

In dealing with non-directed graphs, one sometimes uses a representation derived as the *or* over these planes (*v*). This is equivalent to *IIFC C*.

The incidence matrix *I* has a number of useful properties. For example, *+I* is zero, *+I* gives the difference between the in- and out-degrees of each node, *oI* gives the number of edges followed by the number of nodes, and *x/oI* gives their product. However, all of these are also easily expressed in terms of the connection matrix, and more significant properties of the incidence matrix are seen in its use in electric circuits. For example, if the edges represent components connected between the nodes, and if *v* is the vector of node voltages, then the branch voltages are given by *I+vx*; if *BI* is the vector of branch currents, the vector of node currents is given by *BI+Ix*.

The inverse transformation from incidence matrix to connection matrix is given by:

```
CPI:Dp(-1+1x/D)\epsilon D+(1-1\cdot\cdot\cdot=w)+x-1+1+D+1\Phi pw
```

The *set membership* function *s* yields a boolean array, of the same shape as its left argument, which shows which of its elements belong to the right argument.

3.5 Symbolic Logic

A boolean function of *n* arguments may be represented by a boolean vector of 2^n elements in a variety of ways, including what are sometimes called the *disjunctive*, *conjunctive*, *equivalence*, and *exclusive-disjunctive* forms. The transformation between any pair of these forms may be represented concisely as some 2^n by 2^n matrix formed

by a related inner product, such as $\tau v \cdot \wedge \tau$, where $\tau \cdot \tau$ is the "truth table" formed by the function τ defined by A.2. These matters are treated fully in [11, Ch.7].

4. Identities and Proofs

In this section we will introduce some widely used identities and provide formal proofs for some of them, including Newton's symmetric functions and the associativity of inner product, which are seldom proved formally.

4.1 Dualities in Inner Products

The dualities developed for reduction and scan extend to inner products in an obvious way. If D_F is the dual of F and D_G is the dual of G with respect to a monadic function H with inverse H_I , and if A and B are matrices, then:

$$A \cdot F \cdot G \cdot B \leftrightarrow H_I(H(A) \cdot D_F \cdot D_G \cdot H(B))$$

For example:

$$\begin{aligned} A \vee \cdot \wedge B &\leftrightarrow \sim(\sim A) \wedge \cdot \vee(\sim B) \\ A \wedge \cdot = B &\leftrightarrow \sim(\sim A) \cdot \wedge \sim(\sim B) \\ A \perp \cdot + B &\leftrightarrow \sim(\sim A) \perp \cdot +(\sim B) \end{aligned}$$

The dualities for inner product, reduction, and scan can be used to eliminate many uses of boolean negation from expressions, particularly when used in conjunction with identities of the following form:

$$\begin{aligned} A \wedge (\sim B) &\leftrightarrow A \geq B \\ (\sim A) \wedge B &\leftrightarrow A \leq B \\ (\sim A) \wedge (\sim B) &\leftrightarrow A \sim B \end{aligned}$$

4.2 Partitioning Identities

Partitioning of an array leads to a number of obvious and useful identities. For example:

$$\times/3 \ 1 \ 4 \ 2 \ 6 \leftrightarrow (\times/3 \ 1) \times (\times/4 \ 2 \ 6)$$

More generally, for any associative function F :

$$\begin{aligned} F/V &\leftrightarrow (F/K+V) \cdot F \cdot (F/K+V) \\ F/V, W &\leftrightarrow (F/V) \cdot F \cdot (F/W) \end{aligned}$$

If F is commutative as well as associative, the partitioning need not be limited to prefixes and suffixes, and the partitioning can be made by compression by a boolean vector U :

$$F/V \leftrightarrow (F/U/V) \cdot F \cdot (F/(\sim U)/V)$$

If U is an empty vector ($0=\rho E$), the reduction F/E yields the identity element of the function F , and the identities therefore hold in the limiting cases $0=k$ and $0=v/U$.

Partitioning identities extend to matrices in an obvious way. For example, if V , H , and A are arrays of ranks 1, 2, and 3, respectively, then:

$$\begin{aligned} V \perp \cdot \times M &\leftrightarrow ((K+V) \perp \cdot \times (K, 1+pM)+M)+(K+V) \perp \cdot \times (K, 0)+M \\ (I, J)+A \perp \cdot \times V &\leftrightarrow ((I, J, 0)+A) \perp \cdot \times V \end{aligned} \quad \begin{matrix} D.1 \\ D.2 \end{matrix}$$

4.3 Summarization and Distribution

Consider the definition and use of the following functions:

$$\begin{aligned} N &: (\vee \neq \wedge \omega \circ \cdot = \omega) / \omega \\ S &: (\vee \omega) \circ \cdot = \omega \end{aligned} \quad \begin{matrix} D.3 \\ D.4 \end{matrix}$$

$$\begin{array}{c} A \perp \cdot \times 3 \ 1 \ 4 \ 1 \\ C \perp \cdot \times 10 \ 20 \ 30 \ 40 \ 50 \\ \hline \end{array} \quad \begin{array}{c} N \ A \quad S \ A \quad (S \ A) \perp \cdot \times C \\ \hline 3 \ 1 \ 4 \quad 1 \ 1 \ 0 \ 0 \ 0 \quad 30 \ 80 \ 40 \\ \quad 0 \ 0 \ 1 \ 0 \ 1 \\ \quad 0 \ 0 \ 0 \ 1 \ 0 \end{array}$$

The function S selects from a vector argument its *nub*, that is, the set of distinct elements it contains. The expression $S A$ gives a boolean "summarization matrix" which relates the elements of A to the elements of its nub. If A is a vector of account numbers and c is an associated vector of costs, then the expression $(S A) \perp \cdot \times c$ evaluated above sums or "summarizes" the charges to the several account numbers occurring in A .

Used as postmultiplier, in expressions of the form $H \perp \cdot \times S A$, the summarization matrix can be used to *distribute* results. For example, if F is a function which is costly to evaluate and its argument v has repeated elements, it may be more efficient to apply F only to the nub of v and distribute the results in the manner suggested by the following identity:

$$F \cdot V \leftrightarrow (F \cdot H \cdot V) \perp \cdot \times S \cdot V \quad D.5$$

The order of the elements of $H \cdot v$ is the same as their order in v , and it is sometimes more convenient to use an *ordered* nub and corresponding *ordered* summarization given by:

$$\begin{aligned} QH &: \# \omega [\# \omega] \\ QS &: (QH\omega) \circ \cdot = \omega \end{aligned} \quad \begin{matrix} D.6 \\ D.7 \end{matrix}$$

The identity corresponding to D.5 is:

$$F \cdot V \leftrightarrow (F \cdot QH \cdot V) \perp \cdot \times QS \cdot V \quad D.8$$

The summarization function produces an interesting result when applied to the function T defined by A.2:

$$+\cdot S+\cdot T \cdot N \leftrightarrow (0, 1N)!N$$

In words, the sums of the rows of the summarization matrix of the column sums of the subset matrix of order N is the vector of binomial coefficients of order N .

4.4 Distributivity

The distributivity of one function over another is an important notion in mathematics, and we will now raise the question of representing this in a general way. Since multiplication distributes to the right over addition we have $a \times (b+q) \leftrightarrow ab+aq$, and since it distributes to the left we have $(a+p) \times b \leftrightarrow ab+pb$. These lead to the more general cases:

$$\begin{aligned} (a+p) \times (b+q) &\leftrightarrow ab+aq+pb+pq \\ (a+p) \times (b+q) \times (c+r) &\leftrightarrow abc+abcr+aqc+acr+pbcr+pcr+pcqr \\ (a+p) \times (b+q) \times \dots \times (c+r) &\leftrightarrow ab\dots c+ \dots + pq\dots r \end{aligned}$$

Using the notion that $v \cdot A, B$ and $w \cdot P, Q$ or $v \cdot A, B, C$ and $w \cdot P, Q, R$, etc., the left side can be written simply in terms of reduction as $\times / V \cdot W$. For this case of three elements, the right side can be written as the sum of the products over the columns of the following matrix:

$$\begin{matrix} v[0] & v[0] & v[0] & v[0] & w[0] & w[0] & w[0] \\ v[1] & v[1] & w[1] & w[1] & v[1] & v[1] & w[1] \\ v[2] & w[2] & v[2] & w[2] & v[2] & w[2] & v[2] \end{matrix}$$

The pattern of v 's and w 's above is precisely the pattern of zeros and ones in the matrix $T \cdot \Sigma \cdot v$, and so the products down the columns are given by $(v \cdot \cdot \cdot T) \times (w \cdot \cdot \cdot T)$. Consequently:

$$\times / V \cdot W \leftrightarrow +/ (v \cdot \cdot \cdot T) \times (w \cdot \cdot \cdot T) \quad D.9$$

We will now present a formal inductive proof of D.9, assuming as the induction hypothesis that D.9 is true for all v and w of shape n (that is, $\wedge / n = (\wedge / v, \wedge / w)$) and proving that it holds for shape $n+1$, that is, for $x \cdot v$ and $y \cdot w$, where x and y are arbitrary scalars.

For use in the inductive proof we will first give a recursive definition of the function ζ , equivalent to A.2 and based on the following notion: if $M \cdot \zeta \cdot 2$ is the result of order ζ , then:

$$\begin{matrix} M \\ 0 0 1 1 \\ 0 1 0 1 \\ 0, [1]M & 1, [1]M \\ 0 0 0 0 & 1 1 1 1 \\ 0 0 1 1 & 0 0 1 1 \\ 0 1 0 1 & 0 1 0 1 \\ (0, [1]M), (1, [1]M) \\ 0 0 0 1 1 1 1 \\ 0 0 1 1 0 0 1 1 \\ 0 1 0 1 0 1 0 1 \end{matrix}$$

Thus:

$$T := (0, [1]T), (1, [1]T \cdot \Sigma \cdot \omega - 1); 0 = \omega : 0 \ 1 \omega \quad D.10$$

$$\begin{aligned} & +/ ((C \cdot X, V) \cdot \cdot \cdot Q) \times D \cdot \cdot \cdot Q \cdot \Sigma \cdot (D + Y, W) \\ & +/ ((C \cdot \cdot \cdot Z, U) \times D \cdot \cdot \cdot (Z = 0, [1]T), U = 1, [1]T \cdot \Sigma \cdot \omega \cdot W) \\ & +/ ((C \cdot \cdot \cdot Z), C \cdot \cdot \cdot U) \times (D \cdot \cdot \cdot Z), D \cdot \cdot \cdot U \\ & +/ ((C \cdot \cdot \cdot Z), C \cdot \cdot \cdot U) \times ((Y = 0) \times W \cdot \cdot \cdot T), (Y = 1) \times W \cdot \cdot \cdot T \\ & +/ ((C \cdot \cdot \cdot Z), C \cdot \cdot \cdot U) \times (W \cdot \cdot \cdot T), Y \cdot \cdot \cdot W \cdot \cdot \cdot T \quad Y = 0 \ 1 \leftrightarrow 1, Y \\ & +/ ((X \cdot V \cdot \cdot \cdot T), V \cdot \cdot \cdot T) \times (W \cdot \cdot \cdot T), Y \cdot \cdot \cdot W \cdot \cdot \cdot T \quad Note 2 \\ & +/ ((X \cdot V \cdot \cdot \cdot T) \times W \cdot \cdot \cdot T), (Y \times (V \cdot \cdot \cdot T) \times W \cdot \cdot \cdot T) \quad Note 3 \\ & +/ ((X \cdot V \cdot \cdot \cdot V \cdot W), (Y \cdot \cdot \cdot V \cdot W)) \quad Induction hypothesis \\ & +/ (X, Y) \times X / V \cdot W \quad (X \cdot S), (Y \cdot S) \leftrightarrow (X, Y) \cdot S \\ & \times / (X \cdot Y), (V \cdot W) \quad Definition of \times / \\ & \times / (X, V) \times (Y, W) \quad + distributes over , \end{aligned} \quad D.10$$

Note 1: $M \cdot \times N, P \leftrightarrow (M \cdot \times N), M \cdot \times P$ (partitioning identity on matrices)

Note 2: $V \cdot \times M \leftrightarrow ((1 + V) \cdot \cdot \cdot (1, 1 + \omega M) \cdot M) + (1 + V) \cdot \cdot \cdot 1 \ 0 \ M$ (partitioning identity on matrices and the definition of C , D , Z , and U)

Note 3: $(V, W) \times P, Q \leftrightarrow (V \times P), W \times Q$

To complete the inductive proof we must show that the putative identity D.9 holds for some value of n . If $n=0$, the vectors A and B are empty, and therefore $X \cdot A \leftrightarrow .X$ and $Y \cdot B \leftrightarrow .Y$. Hence the left side becomes $\times / X + Y$, or simply $X + Y$. The right side becomes $+ / (X \cdot \cdot \cdot Q) \times Y \cdot \cdot \cdot Q$, where $\cdot Q$ is the one-rowed matrix $1 \ 0$ and Q is $0 \ 1$. The right side is therefore equivalent to $+ / (X, 1) \times (1, Y)$, or $X + Y$. Similar examination of the case $n=1$ may be found instructive.

4.5 Newton's Symmetric Functions

If x is a scalar and R is any vector, then $\times / X \cdot R$ is a polynomial in x having the roots R . It is therefore equivalent to some polynomial $c \in x$, and assumption of this equivalence implies that c is a function of R . We will now use D.8 and D.9 to derive this function, which is commonly based on Newton's symmetric functions:

$$\begin{aligned} & \times / X \cdot R \\ & \times / X \cdot (-R) \\ & + / (X \cdot \cdot \cdot \cdot \cdot T) \times (-R) \times \cdot \cdot \cdot T \cdot \Sigma \cdot p \cdot R \\ & (X \cdot \cdot \cdot \cdot \cdot T) \cdot \cdot \cdot \cdot \cdot p \cdot (-R) \cdot \cdot \cdot \cdot \cdot T \\ & (X \cdot S \cdot \cdot \cdot \cdot \cdot T) \cdot \cdot \cdot \cdot \cdot p \\ & ((X \cdot Q \cdot S) \cdot \cdot \cdot \cdot \cdot Q \cdot S) \cdot \cdot \cdot \cdot \cdot p \\ & (X \cdot Q \cdot S) \cdot \cdot \cdot \cdot \cdot ((Q \cdot S) \cdot \cdot \cdot \cdot \cdot p) \\ & (X \cdot Q \cdot S) \cdot \cdot \cdot \cdot \cdot ((Q \cdot S) \cdot \cdot \cdot \cdot \cdot p) \\ & ((Q \cdot S) \cdot \cdot \cdot \cdot \cdot p) \cdot \cdot \cdot \cdot \cdot X \\ & ((Q \cdot S) \cdot \cdot \cdot \cdot \cdot p) \cdot \cdot \cdot \cdot \cdot (((-R) \cdot \cdot \cdot \cdot \cdot T \cdot \Sigma \cdot p \cdot R)) \cdot \cdot \cdot \cdot \cdot X \end{aligned} \quad \begin{array}{l} \text{Def of } \cdot \cdot \cdot \\ \text{Note 1} \\ \text{D.8} \\ \text{+ . . is associative} \\ \text{Note 2} \\ \text{B.1 (polynomial)} \\ \text{Defs of } S \\ \text{and } p \end{array}$$

Note 1: If X is a scalar and B is a boolean vector, then $X \cdot \cdot \cdot B \leftrightarrow X \cdot \cdot \cdot / B$.

Note 2: Since T is boolean and has $p \cdot R$ rows, the sums of its columns range from 0 to $p \cdot R$, and their ordered nub is therefore $0, 1 \cdot p \cdot R$.

4.6 Dyadic Transpose

The dyadic transpose, denoted by \circ , is a generalization of monadic transpose which permutes axes of the right argument, and (or) forms "sectors" of the right argument by coalescing certain axes, all as determined by the left argument. We introduce it here as a convenient tool for treating properties of the inner product.

The dyadic transpose will be defined formally in terms of the selection function

$$SF : (. \cdot \omega)[1 + (\rho \cdot \omega) \cdot \alpha - 1]$$

which selects from its right argument the element whose indices are given by its vector left argument, the shape of which must clearly equal the rank of the right argument. The rank of the result of $K \cdot A$ is r / k , and if I is any suitable left argument of the selection $I \cdot SF \cdot K \cdot A$ then:

$$I \cdot SF \cdot K \cdot A \leftrightarrow (I[K]) \cdot SPA \quad D.11$$

For example, if M is a matrix, then $2 \cdot 1 \cdot M \leftrightarrow \cdot M$ and $1 \cdot 1 \cdot M$ is the diagonal of M ; if T is a rank three array, then $1 \cdot 2 \cdot 2 \cdot \circ T$ is a matrix "diagonal section" of T produced by running together the last two axes, and the vector $1 \cdot 1 \cdot 1 \cdot \circ T$ is the principal body diagonal of T .

The following identity will be used in the sequel:

$$J \cdot \& K \cdot A \leftrightarrow (J[K]) \cdot \& A \quad D.12$$

Proof:

$$\begin{aligned} & I \cdot SF \cdot J \cdot \& K \cdot A \\ & (I[J]) \cdot SF \cdot K \cdot A \\ & ((I[J]) \cdot K) \cdot SF \cdot A \\ & (I[(J[K)]) \cdot SF \cdot A \\ & I \cdot SF \cdot (J[K]) \cdot \& A \end{aligned} \quad \begin{array}{l} \text{Definition of } \& \text{ (D.11)} \\ \text{Definition of } \& \\ \text{Indexing is associative} \\ \text{Definition of } \& \end{array}$$

4.7 Inner Products

The following proofs are stated only for matrix arguments and for the particular inner product $\cdot \cdot \cdot$. They are easily extended to arrays of higher rank and to other inner products $F \cdot G$, where F and G need possess only the properties assumed in the proofs for \cdot and \cdot .

The following identity (familiar in mathematics as a sum over the matrices formed by (outer) products of columns of the first argument with corresponding rows of the second argument) will be used in establishing the associativity and distributivity of the inner product:

$$M \cdot N \leftrightarrow +/1 3 3 2 \otimes M \cdot N \quad D.13$$

Proof: $(I, J)SF M \cdot N$ is defined as the sum over v , where $V[K] \leftrightarrow M[I; K] \times N[K; J]$. Similarly,

$$(I, J)SF +/1 3 3 2 \otimes M \cdot N$$

is the sum over the vector v such that

$$W[K] \leftrightarrow (I, J, K)SF 1 3 3 2 \otimes M \cdot N$$

Thus:

$$\begin{aligned} W[K] \\ (I, J, K)SF 1 3 3 2 \otimes M \cdot N \\ (I, J, K)[1 3 3 2]SF M \cdot N \\ (I, J, K)SF M \cdot N \\ M[I; K] \times N[K; J] \\ V[K] \end{aligned} \quad \begin{array}{l} \text{Def of indexing} \\ \text{Def of Outer product} \end{array} \quad D.12$$

Matrix product distributes over addition as follows:

$$M \cdot (N + P) \leftrightarrow (M \cdot N) + (M \cdot P) \quad D.14$$

Proof:

$$\begin{aligned} M \cdot (N + P) \\ +/J 1 3 3 2 \otimes M \cdot N + P \\ +/J (M \cdot N) + (M \cdot P) \\ +/J (J \otimes M \cdot N) + (J \otimes M \cdot P) \\ (+/J \otimes M \cdot N) + (+/J \otimes M \cdot P) \\ (M \cdot N) + (M \cdot P) \end{aligned} \quad \begin{array}{l} \text{D.13} \\ \text{distributes over } + \\ \text{distributes over } + \\ \text{+ is assoc and comm} \end{array} \quad D.13$$

Matrix product is associative as follows:

$$M \cdot (N \cdot P) \leftrightarrow (M \cdot N) \cdot P \quad D.15$$

Proof: We first reduce each of the sides to sums over sections of an outer product, and then compare the sums. Annotation of the second reduction is left to the reader:

$$\begin{aligned} M \cdot (N \cdot P) \\ M \cdot +/1 3 3 2 \otimes N \cdot P \\ +/1 3 3 2 \otimes M \cdot +/1 3 3 2 \otimes N \cdot P \\ +/1 3 3 2 \otimes +/1 3 3 2 \otimes N \cdot P \\ +/1 3 3 2 \otimes +/1 2 3 5 5 4 \otimes M \cdot N \cdot P \\ +/+/1 3 3 2 4 \otimes 1 2 3 5 5 4 \otimes M \cdot N \cdot P \\ +/+/1 3 3 4 4 2 \otimes M \cdot N \cdot P \\ +/+/1 3 3 4 4 2 \otimes (M \cdot N) \cdot P \\ +/+/1 4 4 3 3 2 \otimes (M \cdot N) \cdot P \\ (M \cdot N) \cdot P \\ (+/1 3 3 2 \otimes M \cdot N) \cdot P \\ +/1 3 3 2 \otimes (+/1 3 3 2 \otimes M \cdot N) \cdot P \\ +/1 3 3 2 \otimes +/1 5 5 2 3 4 \otimes (M \cdot N) \cdot P \\ +/+/1 3 3 2 4 \otimes 1 5 5 2 3 4 \otimes (M \cdot N) \cdot P \\ +/+/1 4 4 3 3 2 \otimes (M \cdot N) \cdot P \end{aligned} \quad \begin{array}{l} \text{D.12} \\ \text{D.12} \\ \text{distributes over } + \\ \text{Note 1} \\ \text{Note 2} \\ \text{D.12} \\ \text{x is associative} \\ \text{x is associative and commutative} \end{array}$$

Note 1. $+/M \cdot N \cdot J \otimes A \leftrightarrow +/((1 \otimes M), J \otimes N \otimes A)$

Note 2. $J \otimes +/A \leftrightarrow +/(J, 1 + \Gamma / J) \otimes A$

4.8 Product of Polynomials

The identity B.2 used for the multiplication of polynomials will now be developed formally:

$$\begin{aligned} & (B \cdot E \cdot X) \times (C \cdot E \cdot X) \\ & (+/B \times X \cdot E^{-1} + 1 \otimes B) \times (+/C \times X \cdot E^{-1} + 1 \otimes C) \\ & +/+/B \times X \cdot E \times C \times X \cdot E \\ & +/+/B \cdot E \times C \times (X \cdot E) \times C \times X \cdot E \\ & +/+/B \cdot E \times C \times X \cdot (E \cdot F) \end{aligned} \quad \begin{array}{l} \text{Note 1} \\ \text{Note 2} \\ \text{Note 3} \end{array}$$

Note 1: $(+/V) \times (+/W) \leftrightarrow +/+/V \cdot W \cdot X$ because \cdot distributes over $+$ and $+$ is associative and commutative, or see [12,P21] for a proof.

Note 2: The equivalence of $(P \cdot V) \cdot W \cdot X$ and $(P \cdot V) \times (W \cdot X)$ can be established by examining a typical element of each expression.

Note 3: $(X \cdot I) \times (Y \cdot J) \leftrightarrow X \cdot Y \cdot (I \cdot J)$

The foregoing is the proof presented, in abbreviated form, by Orth [13, p.52], who also defines functions for the composition of polynomials.

4.9 Derivative of a Polynomial

Because of their ability to approximate a host of useful functions, and because they are closed under addition, multiplication, composition, differentiation, and integration, polynomial functions are very attractive for use in introducing the study of calculus. Their treatment in elementary calculus is, however, normally delayed because the derivative of a polynomial is approached indirectly, as indicated in Section 2, through a sequence of more general results.

The following presents a derivation of the derivative of a polynomial directly from the expression for the slope of the secant line through the points X , $E \cdot X$ and $(X + Y)$, $P(X + Y)$:

$$\begin{aligned} & ((C \cdot E \cdot X + Y) - (C \cdot E \cdot X)) \cdot Y \\ & ((C \cdot E \cdot X + Y) - (C \cdot E \cdot X + 0)) \cdot Y \\ & (((C \cdot E \cdot X + Y) - ((0 \cdot Y) + 1 \otimes C)) \cdot E \cdot X) \cdot Y \quad B.6 \\ & (((((Y \cdot J) + 1 \otimes M) \cdot E \cdot X) - ((0 \cdot Y) + 1 \otimes M) \cdot E \cdot X) \cdot Y \quad B.6 \\ & (((((Y \cdot J) + 1 \otimes M) \cdot E \cdot X) - (0 \cdot Y) \cdot E \cdot X) \cdot Y \quad \text{dist over } - \\ & (((((Y \cdot J) + 1 \otimes M) \cdot E \cdot X) + Y \quad + \cdot \text{ dist over } - \\ & (((0, Y \cdot 1 + J) + 1 \otimes M) \cdot E \cdot X) + Y \quad \text{Note 1} \\ & (((Y \cdot 1 + J) + 1 \otimes 1 0 + N) \cdot E \cdot X) + Y \quad D.1 \\ & (((Y \cdot 1 + J) + 1 \otimes (1 0 0 + A) + 1 \otimes C) \cdot E \cdot X) + Y \quad D.2 \\ & (((Y \cdot 1 + J) + 1 \otimes (1 0 0 + A) + 1 \otimes C) \cdot E \cdot X) + Y \quad (Y + A) + Y \leftrightarrow Y + A - 1 \\ & (((Y \cdot 1 + J) + 1 \otimes 1 + pC) + 1 \otimes (1 0 0 + A) + 1 \otimes C) \cdot E \cdot X \quad \text{Def of } J \\ & (((Y \cdot 1 + J) + 1 \otimes 1 + pC) + 1 \otimes (1 0 0 + A) + 1 \otimes C) \cdot E \cdot X \quad D.15 \end{aligned}$$

Note 1: $0 \cdot 0 \leftrightarrow 1 + Y \cdot 0$ and $A \cdot 0 = 0 \cdot 1 + J$

The derivative is the limiting value of the secant slope for y at zero, and the last expression above can be evaluated for this case because if $E^{-1} + 1 + pC$ is the vector of exponents of y , then all elements of E are non-negative. Moreover, $0 \cdot E$ reduces to a 1 followed by zeros, and the inner product with $1 0 0 + A$ therefore reduces to the first plane of $1 0 0 + A$ or, equivalently, the second plane of A .

If $B + J \cdot 1 + 1 + pC$ is the matrix of binomial coefficients, then A is $DS B$ and, from the definition of DS in B.5, the second plane of A is $B \times 1 = -J \cdot 1 - J$, that is, the matrix B with all but the first super-diagonal replaced by zeros. The final expression for the coefficients of the polynomial which is the derivative of the polynomial $C \cdot E \cdot w$ is therefore:

$$((J \cdot 1 + J) \times 1 = -J \cdot 1 - J + 1 + pC) \cdot + \cdot C$$

For example:

```

C + 5 7 11 13
(J..!J)×1=-J..-J+~1+10C
0 1 0 0
0 0 2 0
0 0 0 3
0 0 0 0
((J..!J)×1=-J..-J+~1+10C)+.×C
7 22 39 0

```

Since the superdiagonal of the binomial coefficient matrix $(_{1N})_{0..!N}$ is $(^{-1+(N-1))!}N-1$, or simply $_{N-1}$, the final result is $_{10C+~1+10C}$ in agreement with the earlier derivation.

In concluding the discussion of proofs, we will re-emphasize the fact that all of the statements in the foregoing proofs are executable, and that a computer can therefore be used to identify errors. For example, using the canonical function definition mode [4, p.81], one could define a function F whose statements are the first four statements of the preceding proof as follows:

```

VF
[1] ((C E X+Y)-(C E X))+Y
[2] ((C E X+Y)-(C E X+0))+Y
[3] ((C E X+Y)-((0+J)+.×(A+DS J..-1J+~1+10C)+.×C) E X)+Y
[4] (((Y+J)+.×M) E X)-((0+J)+.×M+A+.×C) E X)+Y
V

```

The statements of the proof may then be executed by assigning values to the variables and executing F as follows:

```

C←5 2 3 1
Y←5
X←3           X←10
F              F
132      66 98 132 174 222 276 336 402 474 552
132      66 96 132 174 222 276 336 402 474 552
132      66 96 132 174 222 276 336 402 474 552
132      66 96 132 174 222 276 336 402 474 552

```

The annotations may also be added as comments between the lines without affecting the execution.

5. Conclusion

The preceding sections have attempted to develop the thesis that the properties of executability and universality associated with programming languages can be combined, in a single language, with the well-known properties of mathematical notation which make it such an effective tool of thought. This is an important question which should receive further attention, regardless of the success or failure of this attempt to develop it in terms of APL.

In particular, I would hope that others would treat the same question using other programming languages and conventional mathematical notation. If these treatments addressed a common set of topics, such as those addressed here, some objective comparisons of languages could be made. Treatments of some of the topics covered here are already available for comparison. For example, Kerner [7] expresses the algorithm C.3 in both ALGOL and conventional mathematical notation.

This concluding section is more general, con-

cerning comparisons with mathematical notation, the problems of introducing notation, extensions to APL which would further enhance its utility, and discussion of the mode of presentation of the earlier sections.

5.1 Comparison with Conventional Mathematical Notation

Any deficiency remarked in mathematical notation can probably be countered by an example of its rectification in some particular branch of mathematics or in some particular publication; comparisons made here are meant to refer to the more general and commonplace use of mathematical notation.

APL is similar to conventional mathematical notation in many important respects: in the use of functions with explicit arguments and explicit results, in the concomitant use of composite expressions which apply functions to the results of other functions, in the provision of graphic symbols for the more commonly used functions, in the use of vectors, matrices, and higher-rank arrays, and in the use of operators which, like the derivative and the convolution operators of mathematics, apply to functions to produce functions.

In the treatment of functions APL differs in providing a precise formal mechanism for the definition of new functions. The direct definition form used in this paper is perhaps most appropriate for purposes of exposition and analysis, but the canonical form referred to in the introduction, and defined in [4, p.81], is often more convenient for other purposes.

In the interpretation of composite expressions APL agrees in the use of parentheses, but differs in eschewing hierarchy so as to treat all functions (user-defined as well as primitive) alike, and in adopting a single rule for the application of both monadic and dyadic functions: the right argument of a function is the value of the entire expression to its right. An important consequence of this rule is that any portion of an expression which is free of parentheses may be read *analytically* from left to right (since the leading function at any stage is the "outer" or overall function to be applied to the result on its right), and *constructively* from right to left (since the rule is easily seen to be equivalent to the rule that *execution* is carried out from right to left).

Although Cajori does not even mention rules for the order of execution in his two-volume history of mathematical notations, it seems reasonable to assume that the motivation for the familiar hierarchy (power before \times and \times before $+$ or $-$) arose from a desire to make polynomials expressible without parentheses. The convenient use of vec-

Fig. 3.

$$\begin{aligned}
 & \sum_{j=1}^n j \cdot 2^{-j} \\
 1 \cdot 2 \cdot 3 + 2 \cdot 3 \cdot 4 + \dots n \text{ terms} & \leftrightarrow \frac{1}{4} n(n+1)(n+2)(n+3) \\
 1 \cdot 2 \cdot 3 \cdot 4 + 2 \cdot 3 \cdot 4 \cdot 5 + \dots n \text{ terms} & \leftrightarrow \frac{1}{5} n(n+1)(n+2)(n+3)(n+4) \\
 & \frac{\left[\frac{x-a}{N}\right]^{-q}}{\Gamma(-q)} \sum_{j=0}^{N-1} \frac{\Gamma(j-q)}{\Gamma(j+1)} f\left(x-j\left[\frac{x-a}{N}\right]\right)
 \end{aligned}$$

tors in expressing polynomials, as in $+/\circ\times\circ\times\circ$, does much to remove this motivation. Moreover, the rule adopted in APL also makes Horner's efficient expression for a polynomial expressible without parentheses:

$+/3 \times 2 \times 5 \times 0 \ 1 \ 2 \ 3 \leftrightarrow 3 \times 4 \times 2 \times 0 \times 5$

In providing graphic symbols for commonly used functions APL goes much farther, and provides symbols for functions (such as the power function) which are implicitly denied symbols in mathematics. This becomes important when operators are introduced; in the preceding sections the inner product $\times\circ\circ$ (which must employ a symbol for power) played an equal role with the ordinary inner product $\circ\circ\circ$. Prohibition of elision of function symbols (such as \times) makes possible the unambiguous use of multi-character names for variables and functions.

In the use of arrays APL is similar to mathematical notation, but more systematic. For example, $v\circ w$ has the same meaning in both, and in APL the definitions for other functions are extended in the same element-by-element manner. In mathematics, however, expressions such as $v\circ w$ and $v\circ w$ are defined differently or not at all.

For example, $v\circ w$ commonly denotes the *vector product* [14, p.308]. It can be expressed in various ways in APL. The definition

$$VP := ((1\Phi a)\times\circ\circ(1\Phi w)) - (\circ\circ(1\Phi a)\times 1\Phi w)$$

provides a convenient basis for an obvious proof that VP is "anticommutative" (that is, $v\circ w \leftrightarrow -w\circ v$), and (using the fact that $\circ\circ x \leftrightarrow 2\Phi x$ for 3-element vectors) for a simple proof that in 3-space v and w are both orthogonal to their vector product, that is, $\circ\circ 0 = v\circ w$ and $\circ\circ 0 = w\circ v$.

APL is also more systematic in the use of operators to produce functions on arrays: reduction provides the equivalent of the sigma and pi notation (in $+/$ and $\times/$) and a host of similar useful cases; outer product extends the outer product of ten-

sor analysis to functions other than \times , and inner product extends ordinary matrix product $(\circ\circ\circ)$ to many cases, such as $\circ\circ\circ$ and $\circ\circ\circ$, for which ad hoc definitions are often made.

The similarities between APL and conventional notation become more apparent when one learns a few rather mechanical substitutions, and the translation of mathematical expressions is instructive. For example, in an expression such as the first shown in Figure 3, one simply substitutes $\circ\circ$ for each occurrence of j and replaces the sigma by $+/$. Thus:

$+/(1\circ N)\times 2\circ\circ\circ\circ\circ 1\circ N$, or $+/\circ J\circ 2\circ\circ\circ\circ\circ J\circ 1\circ N$

Collections such as Jolley's *Summation of Series* [15] provide interesting expressions for such an exercise, particularly if a computer is available for execution of the results. For example, on pages 8 and 9 we have the identities shown in the second and third examples of Figure 3. These would be written as:

$$\begin{aligned}
 +/\circ/(\circ\circ 1\circ N)\circ\circ\circ\circ\circ 1\circ 3 & \leftrightarrow (\circ\circ N\circ 0, 1\circ 3)\circ\circ 4 \\
 +/\circ/(\circ\circ 1\circ N)\circ\circ\circ\circ\circ 1\circ 4 & \leftrightarrow (\circ\circ N\circ 0, 1\circ 4)\circ\circ 5
 \end{aligned}$$

Together these suggest the following identity:

$$+/\circ/(\circ\circ 1\circ N)\circ\circ\circ\circ\circ 1\circ K \leftrightarrow (\circ\circ N\circ 0, 1\circ K)\circ\circ K+1$$

The reader might attempt to restate this general identity (or even the special case where $K=0$) in Jolley's notation.

The last expression of Figure 3 is taken from a treatment of the fractional calculus [16, p.30], and represents an approximation to the q th order derivative of a function f . It would be written as:

$$(S\circ\circ Q)\circ\circ\circ\circ\circ (J\circ J-1\circ Q)\times F X-(J\circ\circ\circ\circ\circ 1\circ N)\times S-(X-A)\circ\circ N$$

The translation to APL is a simple use of $\circ\circ$ as suggested above, combined with a straightforward identity which collapses the several occurrences of the gamma function into a single use of the binomial coefficient function $\circ\circ$, whose domain is, of course, not restricted to integers.

In the foregoing, the parameter q specifies the order of the derivative if positive, and the order of

the integral (from a to x) if negative. Fractional values give fractional derivatives and integrals, and the following function can, by first defining a function r and assigning suitable values to n and a , be used to experiment numerically with the derivatives discussed in [16]:

$$OS := (S - a)^{1/n} / (J! (J - 1 + \alpha) \times \rho \omega - (J + 1 + \alpha) \times S + (\omega - A) \times n)$$

Although much use is made of "formal" manipulation in mathematical notation, truly formal manipulation by explicit algorithms is very difficult. APL is much more tractable in this respect. In Section 2 we saw, for example, that the derivative of the polynomial expression $(\omega \cdot x^{-1 + \alpha})^{1/\alpha}$ is given by $(\omega \cdot x^{-1 + \alpha})^{1/\alpha} \times \frac{1}{\alpha} x^{-1 + \alpha}$, and a set of functions for the formal differentiation of APL expressions given by Orth in his treatment of the calculus [13] occupies less than a page. Other examples of functions for formal manipulation occur in [17, p.347] in the modeling operators for the vector calculus.

Further discussion of the relationship with mathematical notation may be found in [3] and in the paper "Algebra as a Language" [6, p.325].

A final comment on printing, which has always been a serious problem in conventional notation. Although APL does employ certain symbols not yet generally available to publishers, it employs only 88 basic characters, plus some composite characters formed by superposition of pairs of basic characters. Moreover, it makes no demands such as the inferior and superior lines and smaller type fonts used in subscripts and superscripts.

5.2 The Introduction of Notation

At the outset, the ease of introducing notation in context was suggested as a measure of suitability of the notation, and the reader was asked to observe the process of introducing APL. The utility of this measure may well be accepted as a truism, but it is one which requires some clarification.

For one thing, an ad hoc notation which provided exactly the functions needed for some particular topic would be easy to introduce in context. It is necessary to ask further questions concerning the total bulk of notation required, the degree of structure in the notation, and the degree to which notation introduced for a specific purpose proves more generally useful.

Secondly, it is important to distinguish the difficulty of describing and of learning a piece of notation from the difficulty of mastering its implications. For example, learning the rules for computing a matrix product is easy, but a mastery of its implications (such as its associativity, its distributivity over addition, and its ability to represent

linear functions and geometric operations) is a different and much more difficult matter.

Indeed, the very suggestiveness of a notation may make it seem harder to learn because of the many properties it suggests for exploration. For example, the notation $\cdot \cdot \cdot$ for matrix product cannot make the rules for its computation more difficult to learn, since it at least serves as a reminder that the process is an addition of products, but any discussion of the properties of matrix product in terms of this notation cannot help but suggest a host of questions such as: Is $\cdot \cdot \cdot$ associative? Over what does it distribute? Is $B \cdot \cdot \cdot A C \leftrightarrow B(BC) \cdot \cdot \cdot AC$ a valid identity?

5.3 Extensions to APL

In order to ensure that the notation used in this paper is well-defined and widely available on existing computer systems, it has been restricted to current APL as defined in [4] and in the more formal standard published by STAPL, the ACM SIGPLAN Technical Committee on APL [17, p.409]. We will now comment briefly on potential extensions which would increase its convenience for the topics treated here, and enhance its suitability for the treatment of other topics such as ordinary and vector calculus.

One type of extension has already been suggested by showing the execution of an example (roots of a polynomial) on an APL system based on complex numbers. This implies no change in function symbols, although the domain of certain functions will have to be extended. For example, $|x$ will give the magnitude of complex as well as real arguments, $+x$ will give the conjugate of complex arguments as well as the trivial result it now gives for real arguments, and the elementary functions will be appropriately extended, as suggested by the use of \cdot in the cited example. It also implies the possibility of meaningful inclusion of primitive functions for zeros of polynomials and for eigenvalues and eigenvectors of matrices.

A second type also suggested by the earlier sections includes functions defined for particular purposes which show promise of general utility. Examples include the *nub* function $\#$, defined by D.3, and the *summarization* function $\$$, defined by D.4. These and other extensions are discussed in [18]. McDonnell [19, p.240] has proposed generalizations of *and* and *or* to non-booleans so that $A \wedge B$ is the GCD of A and B , and $A \wedge B$ is the LCM. The functions *GCD* and *LCM* defined in Section 3 could then be defined simply by $GCD := \wedge / \wedge$ and $LCM := \wedge / \wedge$.

A more general line of development concerns operators, illustrated in the preceding sections by the reduction, inner-product, and outer-product. Discussions of operators now in APL may be found

in [20] and in [17, p.129], proposed new operators for the vector calculus are discussed in [17, p.47], and others are discussed in [18] and in [17, p.129].

5.4 Mode of Presentation

The treatment in the preceding sections concerned a set of brief topics, with an emphasis on clarity rather than efficiency in the resulting algorithms. Both of these points merit further comment.

The treatment of some more complete topic, of an extent sufficient for, say, a one- or two-term course, provides a somewhat different, and perhaps more realistic, test of a notation. In particular, it provides a better measure of the amount of notation to be introduced in normal course work.

Such treatments of a number of topics in APL are available, including: high school algebra [6], elementary analysis [5], calculus, [13], design of digital systems [21], resistive circuits [10], and crystallography [22]. All of these provide indications of the ease of introducing the notation needed, and one provides comments on experience in its use. Professor Blaauw, in discussing the design of digital systems [21], says that "APL makes it possible to describe what really occurs in a complex system", that "APL is particularly suited to this purpose, since it allows expression at the high architectural level, at the lowest implementation level, and at all levels between", and that "...learning the language pays off (sic) in- and outside the field of computer design".

Users of computers and programming languages are often concerned primarily with the efficiency of execution of algorithms, and might, therefore, summarily dismiss many of the algorithms presented here. Such dismissal would be short-sighted, since a clear statement of an algorithm can usually be used as a basis from which one may easily derive more efficient algorithms. For example, in the function *STEP* of section 3.2, one may significantly increase efficiency by making substitutions of the form $B \otimes M$ for $(B \otimes N) + \times B$, and in expressions using $+/\times X^{-1} + \otimes C$ one may substitute $X \otimes C$ or, adopting an opposite convention for the order of the coefficients, the expression $X \cdot C$.

More complex transformations may also be made. For example, Kerner's method (C.3) results from a rather obvious, though not formally stated, identity. Similarly, the use of the matrix α to represent permutations in the recursive function R used in obtaining the depth first spanning tree (C.4) can be replaced by the possibly more compact use of a list of nodes, substituting indexing for inner products in a rather obvious, though not com-

pletely formal, way. Moreover, such a recursive definition can be transformed into more efficient non-recursive forms.

Finally, any algorithm expressed clearly in terms of arrays can be transformed by simple, though tedious, modifications into perhaps more efficient algorithms employing iteration on scalar elements. For example, the evaluation of $+/\times$ depends upon every element of x and does not admit of much improvement, but evaluation of \vee/\times could stop at the first element equal to 1, and might therefore be improved by an iterative algorithm expressed in terms of indexing.

The practice of first developing a clear and precise definition of a process without regard to efficiency, and then using it as a guide and a test in exploring equivalent processes possessing other characteristics, such as greater efficiency, is very common in mathematics. It is a very fruitful practice which should not be blighted by premature emphasis on efficiency in computer execution.

Measures of efficiency are often unrealistic because they concern counts of "substantive" functions such as multiplication and addition, and ignore the housekeeping (indexing and other selection processes) which is often greatly increased by less straightforward algorithms. Moreover, realistic measures depend strongly on the current design of computers and of language embodiments. For example, because functions on booleans (such as \wedge/\times and \vee/\times) are found to be heavily used in APL, implementers have provided efficient execution of them. Finally, overemphasis of efficiency leads to an unfortunate circularity in design: for reasons of efficiency early programming languages reflected the characteristics of the early computers, and each generation of computers reflects the needs of the programming languages of the preceding generation.

Acknowledgments. I am indebted to my colleague A.D. Falkoff for suggestions which greatly improved the organization of the paper, and to Professor Donald McIntyre for suggestions arising from his reading of a draft.

Appendix A. Summary of Notation

$F\omega$	SCALAR FUNCTIONS $\alpha F\omega$
ω	Conjugate $+$ Plus
$0 - \omega$	Negative $-$ Minus
$(\omega > 0) - \omega < 0$	Signum \times Times
$1 + \omega$	Reciprocal $/$ Divide
$\omega^T - \omega$	Magnitude $ $ Residue
Integer part	Floor P Minimum $(\omega \times \omega < \alpha) + \alpha \times \omega \geq \alpha$
$- - \omega$	Ceiling I Maximum $-(\alpha) - - \omega$
$2.71828... * \omega$	Exponential $*$ Power $\times/\omega \# \alpha$
Inverse of $*$	Natural log \bullet Logarithm $(\# \omega) \# \alpha$
$\times/1 + \omega$	Factorial $:$ Binomial $(1/\omega) \# (\alpha) \times 1/\omega - \alpha$
$3.14159... \times \omega$	Pi times \circ
Boolean: $\vee \vee \sim$ (and, or, not-and, not-or, not)	
Relations: $< \leq = \geq > \neq$ ($\alpha R \omega$ is 1 if relation R holds).	

	Sec.	<i>V</i> ↔ 2 3 5	<i>M</i> ↔ 1 2 3
	Ref.		4 5 6
Integers	1	$i_5 \leftrightarrow 1 2 3 4 5$	
Shape	1	$\rho V \leftrightarrow 3 \rho M \leftrightarrow 2 3 2 3 p \leftrightarrow M$	
Catenation	1	$V, V \leftrightarrow 2 3 5 2 3 5 M, M \leftrightarrow 1 2 3 1 2 3$	4 5 6 4 5 6
Ravel	1	$.M \leftrightarrow 1 2 3 4 5 6$	
Indexing	1	$V[3 1] \leftrightarrow 5 2 M[2; 2] \leftrightarrow 5 M[2; 1] \leftrightarrow 4 5 6$	
Compress	3	$1 0 1 / V \leftrightarrow 2 5 0 1 M \leftrightarrow 4 5 6$	
Take,Drop	1	$2 \leftrightarrow V \leftrightarrow 2 3 2 \leftrightarrow V \leftrightarrow 1 + V \leftrightarrow 3 5$	
Reversal	1	$\phi V \leftrightarrow 5 3 2$	
Rotate	1	$2 \phi V \leftrightarrow 5 2 3 -2 \phi V \leftrightarrow 3 5 2$	
Transpose	1, 4	ω reverses axes $\alpha \omega$ permutes axes	
Grade	3	$\# 3 2 6 2 \leftrightarrow 2 4 1 3 \# 3 2 6 2 \leftrightarrow 3 1 2 4$	
Base value	1	$101V \leftrightarrow 235 V_1 V \leftrightarrow 50$	
&inverse	1	$10 10 10 \tau 235 \leftrightarrow 2 3 5 \tau 50 \leftrightarrow 2 3 5$	
Membership	3	$V \leftrightarrow 3 \leftrightarrow 0 1 0 V \leftrightarrow 5 2 \leftrightarrow 1 0 1$	
Inverse	2, 5	ω is matrix inverse $\alpha \omega \leftrightarrow (\bar{\omega})^+ \times \alpha$	
Reduction	1	$+ / V \leftrightarrow 10 + / M \leftrightarrow 6 15 + / M \leftrightarrow 5 7 9$	
Scan	1	$+ \backslash V \leftrightarrow 2 5 10 + \backslash M \leftrightarrow 2 3 p 1 3 6 4 9 15$	
Inner prod	1	$\times \times$ is matrix product	
Outer prod	1	$0 3 \times . + 1 2 3 \leftrightarrow M$	
Axis	1	$F[I]$ applies F along axis I	

Appendix B. Compiler from Direct to Canonical Form

This compiler has been adapted from [22, p.222]. It will not handle definitions which include α or ω in quotes. It consists of the functions FIX and $F9$, and the character matrices $c9$ and $a9$:

```

FIX
0ρ□FX F9 □

D+P9 E;F;I;K
F+1,(E='ω')o.=5+1)/,E,(φu,ρE)o' Y9 '
F+(,(F='α')o.=5+1)/,F,(φu,ρF)o' X9 '
F+1+pD+(0,+/6,I)+(-3×I)+\I+1:=F)φF,(φ6,ρF)o' '
D+3φC9[1+(1+|α'| $\times$ E),I,0;1,WD[,1,(I+2+F),2]
K+K+2×K<1ΦK+I $\wedge$ K<(1+|α'| $\times$ E)/K+ $\wedge$ |I+E $\in$ A9
F+(0,1+pE)[pD-D,(F,oE)+@0 "-2+Rφ' ',E,[1.5]';'
D+(F+D),[1]F[2] 'a',E
C9
Z9+          012345678
Y929+        9ABCDEPGH
Y929+X9      IJKLMNOPQ
) / 3 -(0=1+.  RSTUVWXYZ
  +0,0ρZ9+     ABCDEEGHI
  JKLMNQPOQR
  STUVWXYZD

```

Example:

```

FIX
FIB:Z,+/~2+Z+FIBω-1:ω=1:1

FIB 15
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610

□CR'FIB'
Z9+P9 Y9;2
+(0=1+,Y9=1)/3
+0,0ρZ9+
Z9+Z,+/~2+Z+FIB Y9-1
*FIB:Z,+/~2+Z+FIBω-1:ω=1:1

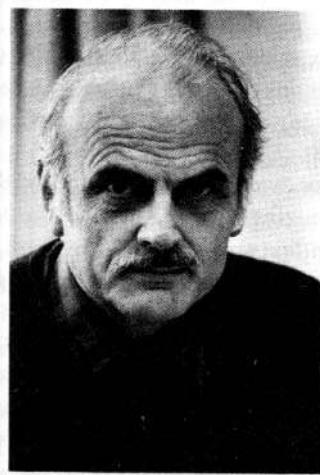
```

References

- Boole, G. *An Investigation of the Laws of Thought*. Dover Publications, N.Y., 1951. Originally published in 1954 by Walton and Maberly, London and by MacMillan and Co., Cambridge. Also available in Volume II of the *Collected Logical Works of George Boole*, Open Court Publishing Co., La Salle, Illinois, 1916.
- Cajori, F. *A History of Mathematical Notations*, Volume II, Open Court Publishing Co., La Salle, Illinois, 1929.
- Falkoff, A.D., and Iverson, K.E. *The Evolution of APL*, *Proceedings of a Conference on the History of Programming Languages*, ACM SIGPLAN, 1978.
- APL Language*, Form No. GC26-3847-4, IBM Corporation.
- Iverson, K.E. *Elementary Analysis*, APL Press, Pleasantville, N.Y., 1976.
- Iverson, K.E. *Algebra: an algorithmic treatment*, APL Press, Pleasantville, N.Y., 1972.
- Kerner, I.O. Ein Gesamtachrittverfahren zur Berechnung der Nullstellen von Polynomen, *Numerische Mathematik*, Vol. 8, 1966, pp. 290-294.
- Beckenbach, E.F., ed. *Applied Combinatorial Mathematics*, John Wiley and Sons, New York, N.Y., 1964.
- Tarjan, R.E. Testing Flow Graph Reducibility, *Journal of Computer and Systems Sciences*, Vol. 9 No. 3, Dec. 1974.
- Spence, R. *Resistive Circuit Theory*, APL Press, Pleasantville, N.Y., 1972.
- Iverson, K.E. *A Programming Language*, John Wiley and Sons, New York, N.Y., 1962.
- Iverson, K.E. *An Introduction to APL for Scientists and Engineers*, APL Press, Pleasantville, N.Y.
- Orth, D.L. *Calculus in a new key*, APL Press, Pleasantville, N.Y., 1976.
- Apostol, T.M. *Mathematical Analysis*, Addison Wesley Publishing Co., Reading, Mass., 1957.
- Jolley, L.B.W. *Summation of Series*, Dover Publications, N.Y., 1961.
- Oldham, K.B., and Spanier, J. *The Fractional Calculus*, Academic Press, N.Y., 1974.
- APL Quote Quad*, Vol. 9, No. 4, June 1979, ACM STAPL.
- Iverson, K.E., *Operators and Functions*, IBM Research Report RC 7091, 1978.
- McDonnell, E.E., A Notation for the GCD and LCM Functions, *APL 75, Proceedings of an APL Conference*, ACM, 1975.
- Iverson, K.E., Operators, *ACM Transactions on Programming Languages And Systems*, October 1979.
- Blaauw, G.A. *Digital System Implementation*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
- McIntyre, D.B., The Architectural Elegance of Crystals Made Clear by APL, *An APL Users Meeting*, I.P. Sharp Associates, Toronto, Canada, 1978.

The 1981 ACM Turing Award Lecture

Delivered at ACM '81, Los Angeles, California, November 9, 1981



The 1981 ACM Turing Award was presented to Edgar F. Codd, an IBM Fellow of the San Jose Research Laboratory, by President Peter Denning on November 9, 1981 at the ACM Annual Conference in Los Angeles, California. It is the Association's foremost award for technical contributions to the computing community.

Codd was selected by the ACM General Technical Achievement Award Committee for his "fundamental and continuing contributions to the theory and practice of database management systems." The originator of the relational model for databases, Codd has made further important contributions in the development of relational algebra, relational calculus, and normalization of relations.

Edgar F. Codd joined IBM in 1949 to prepare programs for the Selective Sequence Electronic Calculator. Since then, his work in computing has encompassed logical design of computers (IBM 701 and Stretch), managing a computer center in Canada, heading the development of one of the first operating systems with a general multiprogramming capability, contributing to the logic of self-reproducing automata, developing high level techniques for software specification, creating and extending the relational approach to database management, and developing an English analyzing and synthesizing subsystem for casual users of relational databases. He is also the author of *Cellular Automata*, an early volume in the ACM Monograph Series.

Codd received his B.A. and M.A. in Mathematics from Oxford University in England, and his M.Sc. and Ph.D. in Computer and Communication Sciences from the University of Michigan. He is a Member of the National Academy of Engineering (USA) and a Fellow of the British Computer Society.

The ACM Turing Award is presented each year in commemoration of A. M. Turing, the English mathematician who made major contributions to the computing sciences.

Relational Database: A Practical Foundation for Productivity

E. F. Codd
IBM San Jose Research Laboratory

It is well known that the growth in demands from end users for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. There are two complementary approaches to attacking this problem (and both approaches are needed): one is to put end users into direct touch with the information stored in computers; the other is to increase the productivity of data processing professionals in the development of application programs. It is less well known that a single technology,

relational database management, provides a practical foundation for both approaches. It is explained why this is so.

While developing this productivity theme, it is noted that the time has come to draw a very sharp line between relational and non-relational database systems, so that the label "relational" will not be used in misleading ways. The key to drawing this line is something called a "relational processing capability."

CR Categories and Subject Descriptors: H.2.0 [Database Management]; General; H.2.1 [Database Management]; Logical Design—*data models*; H.2.4 [Database Management]; Systems

General Terms: Human Factors, Languages

Additional Key Words and Phrases: database, relational database, relational model, data structure, data manipulation, data integrity, productivity

Author's Present Address: E. F. Codd, IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0001-0782/82/0200-0109 \$00.75

1. Introduction

It is generally admitted that there is a productivity crisis in the development of "running code" for commercial and industrial applications. The growth in end user demands for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. In the late sixties and early seventies many people in the computing field hoped that the introduction of database management systems (commonly abbreviated DBMS) would markedly increase the productivity of application programmers by removing many of their problems in handling input and output files. DBMS (along with data dictionaries) appear to have been highly successful as instruments of data control, and they did remove many of the file handling details from the concern of application programmers. Why then have they failed as productivity boosters?

There are three principal reasons:

(1) These systems burdened application programmers with numerous concepts that were irrelevant to their data retrieval and manipulation tasks, forcing them to think and code at a needlessly low level of structural detail (the "owner-member set" of CODASYL DBTG is an outstanding example');

(2) No commands were provided for processing multiple records at a time—in other words, DBMS did not support *set processing* and, as a result, programmers were forced to think and code in terms of iterative loops that were often unnecessary (here we use the word "set" in its traditional mathematical sense, not the linked structure sense of CODASYL DBTG);

(3) The needs of end users for direct interaction with databases, particularly interaction of an unanticipated nature, were inadequately recognized—a query capability was assumed to be something one could add on to a DBMS at some later time.

Looking back at the database management systems of the late sixties, we may readily observe that there was no sharp distinction between the programmer's (logical) view of the data and the (physical) representation of data in storage. Even though what was called the logical level usually provided protection from placement expressed in terms of storage addresses and byte offsets, many storage-oriented concepts were an integral part of this level. The adverse impact on development productivity of requiring programmers to navigate along access paths to

reach the target data (in some cases having to deal directly with the layout of data in storage and in others having to follow pointer chains) was enormous. In addition, it was not possible to make slight changes in the layout in storage without simultaneously having to revise all programs that relied on the previous structure. The introduction of an index might have a similar effect. As a result, far too much manpower was being invested in continual (and avoidable) maintenance of application programs.

Another consequence was that installation of these systems was often agonizingly slow, due to the large amount of time spent in learning about the systems and in planning the organization of the data at both logical and physical levels, prior to database activation. The aim of this preplanning was to "get it right once and for all" so as to avoid the need for subsequent changes in the data description that, in turn, would force coding changes in application programs. Such an objective was, of course, a mirage, even if sound principles for database design had been known at the time (and, of course, they were not).

To show how relational database management systems avoid the three pitfalls cited above, we shall first review the motivation of the relational model and discuss some of its features. We shall then classify systems that are based upon that model. As we proceed, we shall stress application programmer productivity, even though the benefits for end users are just as great, because much has already been said and demonstrated regarding the value of relational database to end users (see [23] and the papers cited therein).

2. Motivation

The most important motivation for the research work that resulted in the relational model was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management (including database design, data retrieval, and data manipulation). We call this the *data independence objective*.

A second objective was to make the model structurally simple, so that all kinds of users and programmers could have a common understanding of the data, and could therefore communicate with one another about the database. We call this the *communicability objective*.

A third objective was to introduce high level language concepts (but not specific syntax) to enable users to express operations upon large chunks of information at a time. This entailed providing a foundation for set-oriented processing (i.e., the ability to express in a single statement the processing of multiple sets of records at a time). We call this the *set-processing objective*.

There were other objectives, such as providing a sound theoretical foundation for database organization and management, but these objectives are less relevant to our present productivity theme.

¹ The crux of the problem with the CODASYL DBTG owner-member set is that it combines into one construct three orthogonal concepts: one-to-many relationship, existence dependency, and a user-visible linked structure to be traversed by application programs. It is the last of these three concepts that places a heavy and unnecessary navigation burden on application programmers. It also presents an insurmountable obstacle for end users.

3. The Relational Model

To satisfy these three objectives, it was necessary to discard all those data structuring concepts (e.g., repeating groups, linked structures) that were not familiar to end users and to take a fresh look at the addressing of data.

Positional concepts have always played a significant role in computer addressing, beginning with plugboard addressing, then absolute numeric addressing, relative numeric addressing, and symbolic addressing with arithmetic properties (e.g., the symbolic address $A + 3$ in assembler language; the address $X(I + 1, J - 2)$ of an element in a Fortran, Algol, or PL/I array named X). In the relational model we replace positional addressing by totally associative addressing. Every datum in a relational database can be uniquely addressed by means of the relation name, primary key value, and attribute name. Associative addressing of this form enables users (yes, and even programmers also!) to leave it to the system to (1) determine the details of placement of a new piece of information that is being inserted into a database and (2) select appropriate access paths when retrieving data.

All information in a relational database is represented by values in tables (even table names appear as character strings in at least one table). Addressing data by value, rather than by position, boosts the productivity of programmers as well as end users (positions of items in sequences are usually subject to change and are not easy for a person to keep track of, especially if the sequences contain many items). Moreover, the fact that programmers and end users all address data in the same way goes a long way to meeting the communicability objective.

The n -ary relation was chosen as the single aggregate structure for the relational model, because with appropriate operators and an appropriate conceptual representation (the table) it satisfies all three of the cited objectives. Note that an n -ary relation is a mathematical set, in which the ordering of rows is immaterial.

Sometimes the following questions arise: Why call it the relational model? Why not call it the tabular model? There are two reasons: (1) At the time the relational model was introduced, many people in data processing felt that a relation (or relationship) among two or more objects must be represented by a linked data structure (so the name was selected to counter this misconception); (2) Tables are at a lower level of abstraction than relations, since they give the impression that positional (array-type) addressing is applicable (which is not true of n -ary relations), and they fail to show that the information content of a table is independent of row order. Nevertheless, even with these minor flaws, tables are the most important conceptual representation of relations, because they are universally understood.

Incidentally, if a data model is to be considered as a serious alternative for the relational model, it too should have a clearly defined conceptual representation for database instances. Such a representation facilitates

thinking about the effects of whatever operations are under consideration. It is a requirement for programmer and end-user productivity. Such a representation is rarely, if ever, discussed in data models that use concepts such as entities and relationships, or in functional data models. Such models frequently do not have any operators either! Nevertheless, they may be useful for certain kinds of data type analysis encountered in the process of establishing a new database, especially in the very early stages of determining a preliminary informal organization. This leads to the question: What is a data model?

A data model is, of course, not just a data structure, as many people seem to think. It is natural that the principal data models are named after their principal structures, but that is not the whole story.

A data model [9] is a combination of at least three components:

(1) A collection of data structure types (the database building blocks);

(2) A collection of operators or rules of inference, which can be applied to any valid instances of the data types listed in (1), to retrieve, derive, or modify data from any parts of those structures in any combinations desired;

(3) A collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of state or both—these rules are general in the sense that they apply to any database using this model (incidentally, they may sometimes be expressed as insert-update-delete rules).

The relational model is a data model in this sense, and was the first such to be defined. We do not propose to give a detailed definition of the relational model here—the original definition appeared in [7], and an improved one in Secs. 2 and 3 of [8]. Its *structural part* consists of domains, relations of assorted degrees (with tables as their principal conceptual representation), attributes, tuples, candidate keys, and primary keys. Under the principal representation, attributes become columns of tables and tuples become rows, but there is no notion of one column succeeding another or of one row succeeding another as far as the database tables are concerned. In other words, the left to right order of columns and the top to bottom order of rows in those tables are arbitrary and irrelevant.

The *manipulative part* of the relational model consists of the algebraic operators (select, project, join, etc.) which transform relations into relations (and hence tables into tables).

The *integrity part* consists of two integrity rules: entity integrity and referential integrity (see [8, 11] for recent developments in this latter area). In any particular application of a data model it may be necessary to impose further (database-specific) integrity constraints, and thereby define a smaller set of consistent database states or changes of state.

In the development of the relational model, there has always been a strong coupling between the structural,

manipulative, and integrity aspects. If the structures are defined alone and separately, their behavioral properties are not pinned down, infinitely many possibilities present themselves, and endless speculation results. It is therefore no surprise that attempts such as those of CODASYL and ANSI to develop data structure definition language (DDL) and data manipulation language (DML) in separate committees have yielded many misunderstandings and incompatibilities.

4. The Relational Processing Capability

The relational model calls not only for relational structures (which can be thought of as tables), but also for a particular kind of set processing called *relational processing*. Relational processing entails treating whole relations as operands. Its primary purpose is loop-avoidance, an absolute requirement for end users to be productive at all, and a clear productivity booster for application programmers.

The SELECT operator (also called RESTRICT) of the relational algebra takes *one* relation (table) as operand and produces a new relation (table) consisting of selected tuples (rows) of the first. The PROJECT operator also transforms *one* relation (table) into a new one, this time however consisting of selected attributes (columns) of the first. The EQUI-JOIN operator takes *two* relations (tables) as operands and produces a third consisting of rows of the first concatenated with rows of the second, but only where specified columns in the first and specified columns in the second have matching values. If redundancy in columns is removed, the operator is called NATURAL JOIN. In what follows, we use the term "join" to refer to either the equi-join or the natural join.

The relational algebra, which includes these and other operators, is intended as a yardstick of power. It is *not* intended to be a standard language, to which all relational systems should adhere. The set-processing objective of the relational model is intended to be met by means of a data sublanguage² having at least the power of the relational algebra *without making use of iteration or recursion statements*.

Much of the derivability power of the relational algebra is obtained from the SELECT, PROJECT, and JOIN operators alone, provided the JOIN is not subject to any implementation restrictions having to do with predefinition of supporting physical access paths. A system has an *unrestricted join capability* if it allows joins to be taken wherein *any* pair of attributes may be matched, providing only that they are defined on the same domain or data type (for our present purpose, it does not matter

whether the domain is syntactic or semantic and it does not matter whether the data type is weak or strong, but see [10] for circumstances in which it does matter).

Occasionally, one finds systems in which join is supported only if the attributes to be matched have the same name or are supported by a certain type of pre-declared access path. Such restrictions significantly impair the power of the system to derive relations from the base relations. These restrictions consequently reduce the system's capability to handle unanticipated queries by end users and reduce the chances for application programmers to avoid coding iterative loops.

Thus, we say that a data sublanguage *L* has a *relational processing capability* if the transformations specified by the SELECT, PROJECT, and unrestricted JOIN operators of the relational algebra can be specified in *L* without resorting to commands for iteration or recursion. For a database management system to be called *relational* it must support:

- (1) Tables without user-visible navigation links between them;
- (2) A data sublanguage with at least this (minimal) relational processing capability.

One consequence of this is that a DBMS that does *not* support relational processing should be considered *non-relational*. Such a system might be more appropriately called *tabular*, providing that it supports tables without user-visible navigation links between tables. This term should replace the term "semi-relational" used in [8], because there is a large difference in implementation complexity between tabular systems, in which the programmer does his own navigation, and relational systems, in which the system does the navigation for him, i.e., the system provides *automatic navigation*.

The definition of relational DBMS given above intentionally permits a lot of latitude in the services provided. For example, it is not required that the full relational algebra be supported, and there is no requirement in regard to support of the two integrity rules of the relational model (entity integrity and referential integrity). Full support by a relational system of these latter two parts of the model justifies calling that system *fully relational* [8]. Although we know of no systems that qualify as fully relational today, some are quite close to qualifying, and no doubt will soon do so.

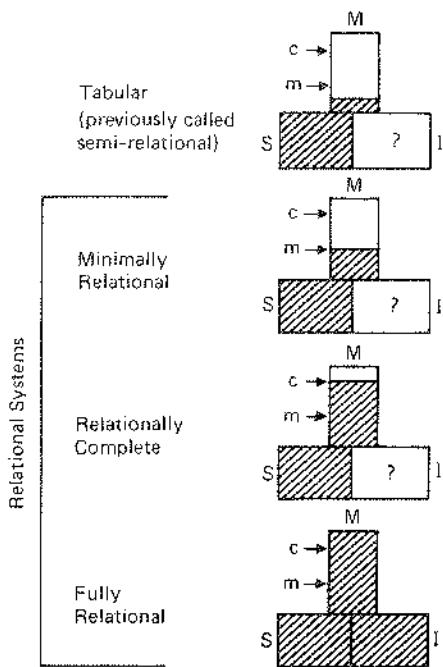
In Fig. 1 we illustrate the distinction between the various kinds of relational and tabular systems. For each class the extent of shading in the S box is intended to show the degree of fidelity of members of that class to the structural requirements of the relational model. A similar remark applies to the M box with respect to the manipulative requirements, and to the I box with respect to the integrity requirements.

m denotes the minimal relational processing capability. *c* denotes relational completeness (a capability corresponding to a two-valued first order predicate logic without nulls). When the manipulation box M is fully shaded, this denotes a capability corresponding to the

²A data sublanguage is a specialized language for database management, supporting at least data definition, data retrieval, insertion, update, and deletion. It need not be computationally complete, and usually is not. In the context of application programming, it is intended to be used in conjunction with one or more programming languages.

Fig. 1. Classification of DBMS.

S = Structural	c = Relational completeness
M = Manipulative	m = Minimal relational processing capability
I = Integrity	



full relational algebra defined in [8] (a three-valued predicate logic with a single kind of null). The question mark in the integrity box for each class except the fully relational is an indication of the present inadequate support for integrity in relational systems. Stronger support for domains and primary keys is needed [10], as well as the kind of facility discussed in [14].

Note that a relational DBMS may package its relational processing capability in any convenient way. For example, in the INGRES system of Relational Technology, Inc., the RETRIEVE statement of QUEL [29] embodies all three operators (select, project, join) in one statement, in such a way that one can obtain the same effect as any one of the operators or any combination of them.

In the definition of the relational model there are several prohibitions. To cite two examples: user-visible navigation links between tables are ruled out, and database information must not be represented (or hidden) in the ordering of tuples within base relations. Our experience is that DBMS designers who have implemented non-relational systems do not readily understand and accept these prohibitions. By contrast, users enthusiastically understand and accept the enhanced ease of learning and ease of use resulting from these prohibitions.

Incidentally, the Relational Task Group of the American National Standards Institute has recently issued a report [4] on the feasibility of developing a standard for relational database systems. This report contains an enlightening analysis of the features of a dozen relational systems, and its authors clearly understand the relational model.

5. The Uniform Relational Property

In order to have wide applicability most relational DBMS have a data sublanguage which can be interfaced with one or more of the commonly used programming languages (e.g., Cobol, Fortran, PL/I, APL). We shall refer to these latter languages as *host languages*. A relational DBMS usually supports at least one end-user oriented data sublanguage—sometimes several, because the needs of these users may vary. Some prefer string languages such as QUEL or SQL [5], while others prefer the screen-oriented two-dimensional data sublanguage of Query-by-Example [33].

Now, some relational systems (e.g., System R [6], INGRES [29]) support a data sublanguage that is usable in two modes: (1) interactively at a terminal and (2) embedded in an application program written in a host language. There are strong arguments for such a *double-mode* data sublanguage:

- (1) With such a language application programmers can separately debug at a terminal the database statements they wish to incorporate in their application programs—people who have used SQL to develop application programs claim that the double-mode feature significantly enhances their productivity;

- (2) Such a language significantly enhances communication among programmers, analysts, end users, database administration staff, etc.;

- (3) Frivolous distinctions between the languages used in these two modes place an unnecessary learning and memory burden on those users who have to work in both modes.

The importance of this feature in productivity suggests that relational DBMS be classified according to whether they possess this feature or not. Accordingly, we call those relational DBMS that support a double-mode sublanguage *uniform relational*. Thus, a uniform relational DBMS supports relational processing at both an end-user interface and at an application programming interface using a data sublanguage common to both interfaces.

The natural term for all other relational DBMS is *non-uniform relational*. An example of a non-uniform relational DBMS is the TANDEM ENCOMPASS [19]. With this system, when retrieving data interactively at a terminal, one uses the relational data sublanguage ENFORM (a language with relational processing capability). When writing a program to retrieve or manipulate data, one uses an extended version of Cobol (a language that does not possess the relational processing capability). Common to both levels of use are the structures: tables without user-visible navigation links between them.

A question that immediately arises is this: how can a data sublanguage with relational processing capability be interfaced with a language such as Cobol or PL/I that can handle data one record at a time only (i.e., that is incapable of treating a set of records as a single operand)? To solve this problem we must separate the following

two actions from one another: (1) definition of the relation to be derived; (2) presentation of the derived relation to the host language program.

One solution (adopted in the Peterlee Relational Test Vehicle [31]) is to cast a derived relation in the form of a file that can be read record-by-record by means of host language statements. In this case delivery of records is delegated to the file system used by the pertinent host language.

Another solution (adopted by System R) is to keep the delivery of records under the control of data sublanguage statements and, hence, under the control of the relational DBMS optimizer. A query statement Q of SQL (the data sublanguage of System R) may be embedded in a host language program, using the following kind of phrase (for expository reasons, the syntax is not exactly that of SQL)

DECLARE C CURSOR FOR Q

where C stands for any name chosen by the programmer. Such a statement associates a *cursor* named C with the defining expression Q. Tuples from the derived relation defined by Q are presented to the program one at a time by means of the named cursor. Each time a FETCH per this cursor is executed, the system delivers another tuple from the derived relation. The order of delivery is system-determined unless the SQL statement Q defining the derived relation contains an ORDER BY clause.

It is important to note that in advancing a cursor over a derived relation the programmer is *not* engaging in navigation to some target data. The derived relation is itself the target data! It is the DBMS that determines whether the derived relation should be materialized *en bloc* prior to the cursor-controlled scan or materialized piecemeal during the scan. In either case, it is the system (not the programmer) that selects the access paths by which the derived data is to be generated. This takes a significant burden off the programmer's shoulders, thereby increasing his productivity.

6. Skepticism About Relational Systems

There has been no shortage of skepticism concerning the practicality of the relational approach to database management. Much of this skepticism stems from a lack of understanding, some from a fear of the numerous theoretical investigations that are based on the relational model [1, 2, 15, 16, 24]. Instead of welcoming a theoretical foundation as providing soundness, the attitude seems to be: if it's theoretical, it cannot be practical. The absence of a theoretical foundation for almost all non-relational DBMS is the prime cause of their *ungepotchket* quality. (This is a Yiddish word, one of whose meanings is patched up.)

On the other hand, it seems reasonable to pose the following two questions:

- (1) Can a relational system provide the range of ser-

vices that we have grown to expect from other DBMS?

(2) If (1) is answered affirmatively, can such a system perform as well as non-relational DBMS?³

We look at each of these in turn.

6.1 Range of Services

A full-scale DBMS provides the following capabilities:

- data storage, retrieval, and update;
- a user-accessible catalog for data description;
- transaction support to ensure that all or none of a sequence of database changes are reflected in the pertinent databases (see [17] for an up-to-date summary of transaction technology);
- recovery services in case of failure (system, media, or program);
- concurrency control services to ensure that concurrent transactions behave the same way as if run in some sequential order;
- authorization services to ensure that all access to and manipulation of data be in accordance with specified constraints on users and programs [18];
- integration with support for data communication;
- integrity services to ensure that database states and changes of state conform to specified rules.

Certain relational prototypes developed in the early seventies fell far short of providing all these services (possibly for good reasons). Now, however, several relational systems are available as software products and provide all these services with the exception of the last. Present versions of these products are admittedly weak in the provision of integrity services, but this is rapidly being remedied [10].

Some relational DBMS actually provide more complete data services than the non-relational systems. Three examples follow.

As a first example, relational DBMS support the extraction of all meaningful relations from a database, whereas non-relational systems support extraction only where there exist statically predefined access paths.

As a second example of the additional services provided by some relational systems, consider views. A *view* is a virtual relation (table) defined by means of an expression or sequence of commands. Although not directly supported by actual data, a view appears to a user as if it were an additional base table kept up-to-date and in a state of integrity with the other base tables. Views are useful for permitting application programs and users at terminals to interact with constant view structures, even when the base tables themselves are undergoing structural changes at the *logical* level (providing that the pertinent views are still definable from the new base tables). They are also useful in restricting the scope of

³ One should bear in mind that the non-relational ones always employ comparatively low level data sublanguages for application programming.

access of programs and users. Non-relational systems either do not support views at all or else support much more primitive counterparts, such as the CODASYL subschema.

As a third example, some systems (e.g., SQL/DS [28] and its prototype predecessor System R) permit a variety of changes to be made to the logical and physical organization of the data dynamically—while transactions are in progress. These changes rarely require application programs to be recoded. Thus, there is less of a program maintenance burden, leaving programmers to be more productive doing development rather than maintenance. This capability is made possible in SQL/DS by the fact that the system has complete control over access path selection.

In non-relational systems such changes would normally require all other database activities including transactions in progress to be brought to a halt. The database then remains out of action until the organizational changes are completed and any necessary recompiling done.

6.2 Performance

Naturally, people would hesitate to use relational systems if these systems were sluggish in performance. All too often, erroneous conclusions are drawn about the performance of relational systems by comparing the time it might take for one of these systems to execute a complex transaction with the time a non-relational system might take to execute an extremely simple transaction. To arrive at a fair performance comparison, one must compare these systems on the same tasks or applications. We shall present arguments to show why relational systems should be able to compete successfully with non-relational systems.

Good performance is determined by two factors: (1) the system must support performance-oriented physical data structures; (2) high-level language requests for data must be compiled into lower-level code sequences at least as good as the average application programmer can produce by hand.

The first step in the argument is that a program written in a Cobol-level language can be made to perform efficiently on large databases containing production data structured in tabular form with no user-visible navigation links between them. This step in the argument is supported by the following information [19]: as of August 1981, Tandem Computer Corp. had manufactured and installed 760 systems; of these, over 700 were making use of the Tandem ENCOMPASS relational database management system to support databases containing production data. Tandem has committed its own manufacturing database to the care of ENCOMPASS. ENCOMPASS does not support links between the database tables, either user-visible (navigation) links or user-invisible (access method) links.

In the second step of the argument, suppose we take the application programs in the above-cited installations

and replace the database retrieval and manipulation statements by statements in a database sublanguage with a relational processing capability (e.g., SQL). Clearly, to obtain good performance with such a high level language, it is essential that it be compiled into object code (instead of being interpreted), and it is essential that that object code be efficient.

Compilation is used in System R and its product version SQL/DS. In 1976 Raymond Lorie developed an ingenious pre- and post-compiling scheme for coping with dynamic changes in access paths [21]. It also copes with early (and hence efficient) authorization and integrity checking (the latter, however, is not yet implemented). This scheme calls for compiling in a rather special way the SQL statements embedded in a host language program. This compilation step transforms the SQL statements into appropriate CALLs within the source program together with access modules containing object code. These modules are then stored in the database for later use at runtime. The code in these access modules is generated by the system so as to optimize the sequencing of the major operations and the selection of access paths to provide runtime efficiency. After this pre-compilation step, the application program is compiled by a regular compiler for the pertinent host language. If at any subsequent time one or more of the access paths is removed and an attempt is made to run the program, enough source information has been retained in the access module to enable the system to re-compile a new access module that exploits the now existing access paths *without requiring a re-compilation of the application program*.

Incidentally, the same data sublanguage compiler is used on ad hoc queries submitted interactively from a terminal and also on queries that are dynamically generated during the execution of a program (e.g., from parameters submitted interactively). Immediately after compilation, such queries are executed and, with the exception of the simplest of queries, the performance is better than that of an interpreter.

The generation of access modules (whether at the initial compiling or re-compiling stage) entails a quite sophisticated optimization scheme [27], which makes use of system-maintained statistics that would not normally be within the programmer's knowledge. Thus, only on the simplest of all transactions would it be possible for an average application programmer to compete with this optimizer in generation of efficient code. Any attempts to compete are bound to reduce the programmer's productivity. Thus, the price paid for extra compile-time overhead would seem to be well worth paying.

Assuming non-linked tabular structures in both cases, we can expect SQL/DS to generate code comparable with average hand-written code in many simple cases, and superior in many complex cases. Many commercial transactions are extremely simple. For example, one may need to look up a record for a particular railroad wagon to find out where it is or find the balance in someone's

savings account. If suitably fast access paths are supported (e.g., hashing), there is no reason why a high-level language such as SQL, QUEL, or QBE should result in less efficient runtime code for these simple transactions than a lower level language, even though such transactions make little use of the optimizing capability of the high-level data sublanguage compiler.

7. Future Directions

If we are to use relational database as a foundation for productivity, we need to know what sort of developments may lie ahead for relational systems.

Let us deal with near-term developments first. In some relational systems stronger support is needed for domains and primary keys per suggestions in [10]. As already noted, all relational systems need upgrading with regard to automatic adherence to integrity constraints. Existing constraints on updating join-type views need to be relaxed (where theoretically possible), and progress is being made on this problem [20]. Support for outer joins is needed.

Marked improvements are being made in optimizing technology, so we may reasonably expect further improvements in performance. In certain products, such as the ICL CAPS [22] and the Britton-Lee IDM500 [13], special hardware support has been implemented. Special hardware may help performance in certain types of applications. However, in the majority of applications dealing with formatted databases, software-implemented relational systems can compete in performance with software-implemented non-relational systems.

At present, most relational systems do not provide any special support for engineering and scientific databases. Such support, including interfacing with Fortran, is clearly needed and can be expected.

Catalogs in relational systems already consist of additional relations that can be interrogated just like the rest of the database using the same query language. A natural development that can and should be swiftly put in place is the expansion of these catalogs into full-fledged active dictionaries to provide additional on-line data control.

Finally, in the near term, we may expect database design aids suited for use with relational systems both at the logical and physical levels.

In the longer term we may expect support for relational databases distributed over a communications network [25, 30, 32] and managed in such a way that application programs and interactive users can manipulate the data (1) as if all of it were stored at the local node—*location transparency*—and (2) as if no data were replicated anywhere—*replication transparency*. All three of the projects cited above are based on the relational model. One important reason for this is that relational databases offer great decomposition flexibility when planning how a database is to be distributed over a

network of computer systems, and great recombination power for dynamic combination of decentralized information. By contrast, CODASYL DBTG databases are very difficult to decompose and recompose due to the entanglement of the owner-member navigation links. This property makes the CODASYL approach extremely difficult to adapt to a distributed database environment and may well prove to be its downfall. A second reason for use of the relational model is that it offers concise high level data sublanguages for transmitting requests for data from node to node.

The ongoing work in extending the relational model to capture in a formal way more meaning of the data can be expected to lead to the incorporation of this meaning in the database catalog in order to factor it out of application programs and make these programs even more concise and simple. Here, we are, of course, talking about meaning that is represented in such a way that the system can understand it and act upon it.

Improved theories are being developed for handling missing data and inapplicable data (see for example [3]). This work should yield improved treatment of null values.

As it stands today, relational database is best suited to data with a rather regular or homogeneous structure. Can we retain the advantages of the relational approach while handling heterogeneous data also? Such data may include images, text, and miscellaneous facts. An affirmative answer is expected, and some research is in progress on this subject, but more is needed.

Considerable research is needed to achieve a rapprochement between database languages and programming languages. Pascal/R [26] is a good example of work in this direction. Ongoing investigations focus on the incorporation of abstract data types into database languages on the one hand [12] and relational processing into programming languages on the other.

8. Conclusions

We have presented a series of arguments to support the claim that relational database technology offers dramatic improvements in productivity both for end users and for application programmers. The arguments center on the data independence, structural simplicity, and relational processing defined in the relational model and implemented in relational database management systems. All three of these features simplify the task of developing application programs and the formulation of queries and updates to be submitted from a terminal. In addition, the first feature tends to keep programs viable in the face of organizational and descriptive changes in the database and therefore reduces the effort that is normally diverted into the maintenance of programs.

Why, then, does the title of this paper suggest that relational database provides only a foundation for improved productivity and not the total solution? The

reason is simple: relational database deals only with the shared data component of application programs and end-user interactions. There are numerous complementary technologies that may help with other components or aspects, for example, programming languages that support relational processing and improved checking of data types, improved editors that understand more of the language being used, etc. We use the term "foundation," because interaction with shared data (whether by program or via terminal) represents the core of so much data processing activity.

The practicality of the relational approach has been proven by the test and production installations that are already in operation. Accordingly, with relational systems we can now look forward to the productivity boost that we all hoped DBMS would provide in the first place.

Acknowledgments. I would like to express my indebtedness to the System R development team at IBM Research, San Jose for developing a full-scale, uniform relational prototype that entailed numerous language and system innovations; to the development team at the IBM Laboratory, Endicott, N.Y. for the professional way in which they converted System R into product form; to the various teams at universities, hardware manufacturers, software firms, and user installations, who designed and implemented working relational systems; to the QBE team at IBM Yorktown Heights, N.Y.; to the PRTV team at the IBM Scientific Centre in England; and to the numerous contributors to database theory who have used the relational model as a cornerstone. A special acknowledgement is due to the very few colleagues who saw something worth supporting in the early stages, particularly, Chris Date and Sharon Weinberg. Finally, it was Sharon Weinberg who suggested the theme of this paper.

Received 10/81; revised and accepted 12/81

References

1. Beeri, C., Bernstein, P., Goodman, N. A sophisticate's introduction to database normalization theory. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept. 1978.
2. Bernstein, P.A., Goodman, N., Lai, M-Y. Laying phantoms to rest. Report TR-03-81, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., 1981.
3. Biskup, J.A. A formal approach to null values in database relations. *Proc. Workshop on Formal Bases for Data Bases*, Toulouse, France, Dec. 1979; published in [16] (see below) pp 299-342.
4. Brodie, M. and Schmidt, J. (Eds), Report of the ANSI Relational Task Group, (to be published ACM SIGMOD Record).
5. Chamberlin, D.D., et al. SEQUEL2: A unified approach to data definition, manipulation, and control. *IBM J. Res. & Dev.*, 20, 6, (Nov. 1976) 560-565.
6. Chamberlin, D.D., et al. A history and evaluation of system R. *Comm. ACM*, 24, 10, (Oct. 1981) 632-646.
7. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM*, 13, 6, (June 1970) 377-387.
8. Codd, E.F. Extending the database relational model to capture more meaning. *ACM TODS*, 4, 4, (Dec. 1979) 397-434.
9. Codd, E.F. Data models in database management. *ACM SIGMOD Record*, 11, 2, (Feb. 1981) 112-114.
10. Codd, E.F. The capabilities of relational database management systems. *Proc. Convención Informática Latina*, Barcelona, Spain, June 5-12, 1981, pp 13-26; also available as Report 3132, IBM Research Lab., San Jose, Calif.
11. Date, C.J. Referential integrity. *Proc. Very Large Data Bases*, Cannes, France, September 9-11, 1981, pp 2-12.
12. Ehrig, H., and Weber, H. Algebraic specification schemes for data base systems. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept 13-15, 1978, 427-440.
13. Epstein, R., and Hawthorne, P. Design decisions for the intelligent database machine. *Proc. NCC 1980, AFIPS, Vol. 49, May 1980*, pp 237-241.
14. Eswaran, K.P., and Chamberlin, D.D. Functional specifications of a subsystem for database integrity. *Proc. Very Large Data Bases*, Framingham, Mass., Sept. 1975, pp 48-68.
15. Fagin, R. Horn clauses and database dependencies. *Proc. 1980 ACM SIGACT Symp. on Theory of Computing*, Los Angeles, CA, pp 123-134.
16. Gallaire, H., Minker, J., and Nicolas, J.M. *Advances in Data Base Theory*. Vol 1, Plenum Press, New York, 1981.
17. Gray, J. The transaction concept: virtues and limitations. *Proc. Very Large Data Bases*, Cannes, France, September 9-11, 1981, pp 144-154.
18. Griffiths, P.G., and Wade, B.W. An authorization mechanism for a relational database system. *ACM TODS*, 1, 3, (Sept 1976) 242-255.
19. Held, G. ENCOMPASS: A relational data manager. *Data Base/81*, Western Institute of Computer Science, Univ. of Santa Clara, Santa Clara, Calif., August 24-28, 1981.
20. Keller, A.M. Updates to relational databases through views involving joins. Report RJ3282, IBM Research Laboratory, San Jose, Calif., October 27, 1981.
21. Lorie, R.A., and Nilsson, J.F. An access specification language for a relational data base system. *IBM J. Res. & Dev.*, 23, 3, (May 1979) 286-298.
22. Maller, V.A.J. The content addressable file store—CAFS. *ICL Technical J.*, 1, 3, (Nov. 1979) 265-279.
23. Reisner, P. Human factors studies of database query languages: A survey and assessment. *ACM Computing Surveys*, 13, 1, (March 1981) 13-31.
24. Rissanen, J. Theory of relations for databases—A tutorial survey. *Proc. Symp. on Mathematical Foundations of Computer Science*, Zakopane, Poland, September 1978, Lecture Notes in Computer Science, No. 64, Springer Verlag, New York, 1978.
25. Rothnie, J.B., Jr. et al. Introduction to a system for distributed databases (SDD-1). *ACM TODS*, 5, 1, (March 1980) 1-17.
26. Schmidt, J.W. Some high level language constructs for data of type relation. *ACM TODS*, 2, 3, (Sept 1977) 247-261.
27. Selinger, P.G., et al. Access path selection in a relational database system. *Proc. 1979 ACM SIGMOD International Conference on Management of Data*, Boston, MA, May 1979, pp 23-34.
28. ———, SQL/Data system for VSE: A relational data system for application development. IBM Corp. Data Processing Division, White Plains, N.Y., G320-6590, Feb 1981.
29. Stonebraker, M.R., et al. The design and implementation of INGRES. *ACM TODS*, 1, 3, (Sept. 1976) 189-222.
30. Stonebraker, M.R., and Neuhold, E.J. A distributed data base version of INGRES. *Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks*, Lawrence-Berkeley Lab., Berkeley, Calif., May 1977, pp 19-36.
31. Todd, S.J.P. The Peterlee relational test vehicle—A system overview. *IBM Systems J.*, 15, 4, 1976, 285-308.
32. Williams, R. et al. R*: An overview of the architecture. Report RJ3325, IBM Research Laboratory, San Jose, Calif., October 27, 1981.
33. Zloof, M.M. Query by example. *Proc. NCC, AFIPS Vol 44, May 1975*, pp 431-438.

AN OVERVIEW OF COMPUTATIONAL COMPLEXITY

STEPHEN A. COOK University of Toronto

This is the second Turing Award lecture on Computational Complexity. The first was given by Michael Rabin in 1976.¹ In reading Rabin's excellent article [62] now, one of the things that strikes me is how much activity there has been in the field since. In this brief overview I want to mention what to me are the most important and interesting results since the subject began in about 1960. In such a large field the choice of topics is inevitably somewhat personal; however, I hope to include papers which, by any standards, are fundamental.

1. EARLY PAPERS

The prehistory of the subject goes back, appropriately, to Alan Turing. In his 1937 paper, *On computable numbers with an application to the Entscheidungs problem* [85], Turing introduced his famous Turing machine, which provided the most convincing formalization (up to that time) of the notion of an effectively (or algorithmically) computable function. Once this notion was pinned down precisely, impossibility proofs for computers were possible. In the same paper Turing proved that no algorithm (i.e., Turing machine) could, upon being given an arbitrary formula of the predicate calculus, decide, in a finite number of steps, whether that formula was satisfiable.

After the theory explaining which problems can and cannot be solved by computer was well developed, it was natural to ask about the relative computational difficulty of computable functions. This is the subject matter of computational complexity. Rabin [59, 60] was one of the first persons (1960) to address this general question explicitly: what does it mean to say that f is more difficult to compute than g ? Rabin suggested an axiomatic framework that provided the basis for the abstract complexity theory developed by Blum [6] and others.

A second early (1965) influential paper was *On the computational complexity of algorithms* by J. Hartmanis and R. E. Stearns [37].² This paper was widely read and gave the field its title. The important notion of complexity measure defined by the computation time on multitape Turing machines was introduced, and hierarchy theorems were proved. The paper also posed an intriguing question that is still open today. Is

ABSTRACT: An historical overview of computational complexity is presented. Emphasis is on the fundamental issues of defining the intrinsic computational complexity of a problem and proving upper and lower bounds on the complexity of problems. Probabilistic and parallel computation are discussed.

Author's Present Address:
Stephen A. Cook,
Department of Computer
Science,
University of Toronto,
Toronto, Canada M5S 1A7.
Permission to copy without fee
all or part of this material is
granted provided that the
copies are not made or
distributed for direct
commercial advantage, the
copyright notice and the
name of the publication and its
appear, and notice is given
copying is by permission of
Association for Computing
Machinery. To copy otherwise,
to republish, requires a fee
and/or specific permission.
1983 ACM 0001-0782/83/
0600-0401 75c.

¹ Michael Rabin and Dana Scott shared the Turing Award in 1976.

² See Hartmanis [36] for some interesting reminiscences.

any irrational algebraic number (such as $\sqrt{2}$) computable in real time, that is, is there a Turing machine that prints out the decimal expansion of the number at the rate of one digit per 100 steps forever.

A third founding paper [1965] was *The intrinsic computational difficulty of functions* by Alan Cobham [15]. Cobham emphasized the word "intrinsic," that is, he was interested in a machine-independent theory. He asked whether multiplication is harder than addition, and believed that the question could not be answered until the theory was properly developed. Cobham also defined and characterized the important class of functions he called \mathcal{A} : those functions on the natural numbers computable in time bounded by a polynomial in the decimal length of the input.

Three other papers that influenced the above authors as well as other complexity workers (including myself) are Yamada [91], Bennett [4], and Ritchie [66]. It is interesting to note that Rabin, Stearns, Bennett, and Ritchie were all students at Princeton at roughly the same time.

2. EARLY ISSUES AND CONCEPTS

Several of the early authors were concerned with the question: What is the right complexity measure? Most mentioned computation time or space as obvious choices, but were not convinced that these were the only or the right ones. For example, Cobham [15] suggested "... some measure related to the physical notion of work [may] lead to the most satisfactory analysis." Rabin [60] introduced axioms which a complexity measure should satisfy. With the perspective of 20 years experience, I now think it is clear that time and space—especially time—are certainly among the most important complexity measures. It seems that the first figure of merit given to evaluate the efficiency of an algorithm is its running time. However, more recently it is becoming clear that parallel time and hardware size are important complexity measures too (see Section 6).

Another important complexity measure that goes back in some form at least to Shannon [74] (1949) is Boolean circuit (or combinational) complexity. Here it is convenient to assume that the function f in question takes finite bit strings into finite bit strings, and the complexity $C(n)$ of f is the size of the smallest Boolean circuit that computes f for all inputs of length n . This very natural measure is closely related to computation time (see [57a], [57b], [68b]), and has a well developed theory in its own right (see Savage [68a]).

Another question raised by Cobham [15] is what constitutes a "step" in a computation. This amounts to asking what is the right computer model for measuring the computation time of an algorithm. Multitape Turing machines are commonly used in the literature, but they have artificial restrictions from the point of view of efficient implementation of algorithms. For example, there is no compelling reason why the storage media should be linear tapes. Why not planar arrays or trees? Why not allow a random access memory?

In fact, quite a few computer models have been proposed since 1960. Since real computers have random access memories, it seems natural to allow these in the model. But just how to do this becomes a tricky question. If the machine can store integers in one step some bound must be placed on their size. (If the number 2 is squared 100 times the result has 2^{100} bits, which could not be stored in all the world's existing storage media.) I proposed charged RAM's in [19], in which a cost (number of steps) of about $\log|x|$ is charged every time a number x is stored or retrieved. This works but is not completely convincing. A more popular random access model is the one used by Aho, Hopcroft, and Ullman in [3], in which

each operation involving an integer has unit cost, but integers are not allowed to become unreasonably large (for example, their magnitude might be bounded by some fixed polynomial in the size of the input). Probably the most mathematically satisfying model is Schönhage's storage modification machine [69], which can be viewed either as a Turing machine that builds its own storage structure or as a unit cost RAM that can only copy, add or subtract one, or store or retrieve in one step. Schönhage's machine is a slight generalization of the Kolmogorov-Uspenski machine proposed much earlier [46] (1958), and seems to me to represent the most general machine that could possibly be construed as doing a bounded amount of work in one step. The trouble is that it probably is a little too powerful. (See Section 3 under "large number multiplication".)

Returning to Cobham's question "what is a step," I think what has become clear in the last 20 years is that there is no single clear answer. Fortunately, the competing computer models are not wildly different in computation time. In general, each can simulate any other by at most squaring the computation time (some of the first arguments to this effect are in [37]). Among the leading random access models, there is only a factor of log computation time in question.

This leads to the final important concept developed by 1965—the identification of the class of problems solvable in time bounded by a polynomial in the length of the input. The distinction between polynomial time and exponential time algorithms was made as early as 1953 by von Neumann [90]. However, the class was not defined formally and studied until Cobham [15] introduced the class \mathcal{A} of functions in 1964 (see Section 1). Cobham pointed out that the class was well defined, independent of which computer model was chosen, and gave it a characterization in the spirit of recursive function theory. The idea that polynomial time computability roughly corresponds to tractability was first expressed in print by Edmonds [27], who called polynomial time algorithms "good algorithms." The now standard notation P for the class of polynomial time recognizable sets of strings was introduced later by Karp [42].

The identification of P with the tractable (or feasible) problems has been generally accepted in the field since the early 1970's. It is not immediately obvious why this should be true, since an algorithm whose running time is the polynomial n^{ω} is surely not feasible, and conversely, one whose running time is the exponential $2^{\omega n^{\omega}}$ is feasible in practice. It seems to be an empirical fact, however, that naturally arising problems do not have optimal algorithms with such running times.³ The most notable practical algorithm that has an exponential worst case running time is the simplex algorithm for linear programming. Smale [75, 76] attempts to explain this by showing that, in some sense, the average running time is fast, but it is also important to note that Khachian [43] showed that linear programming is in P using another algorithm. Thus, our general thesis, that P equals the feasible problems, is not violated.

3. UPPER BOUNDS ON TIME

A good part of computer science research consists of designing and analyzing enormous numbers of efficient algorithms. The important algorithms (from the point of view of computational complexity) must be special in some way; they generally supply a surprisingly fast way of solving a simple or important problem. Below I list some of the more interesting ones invented since 1960. (As an aside, it is interesting to speculate on what are the all time most important algorithms. Surely

³ See [31], pp. 6-9 for a discussion of this.

the arithmetic operations +, -, *, and : on decimal numbers are basic. After that, I suggest fast sorting and searching, Gaussian elimination, the Euclidean algorithm, and the simplex algorithm as candidates.)

The parameter n refers to the size of the input, and the time bounds are the worst case time bounds and apply to a multitape Turing machine (or any reasonable random access machine) except where noted.

(1) **The Fast Fourier Transform** [23], requiring $O(n \log n)$ arithmetic operations, is one of the most used algorithms in scientific computing.

(2) **Large number multiplication.** The elementary school method requires $O(n^2)$ bit operations to multiply two n digit numbers. In 1962 Karatsuba and Ofman [41] published a method requiring only $O(n^{1.5})$ steps.

Shortly after that Toom [84] showed how to construct Boolean circuits of size $O(n^{1+\epsilon})$ for arbitrarily small $\epsilon > 0$ in order to carry out the multiplication. I was a graduate student at Harvard at the time, and inspired by Cobham's question "Is multiplication harder than addition?" I was naively trying to prove that multiplication requires $\Omega(n^2)$ steps on a multitape Turing machine. Toom's paper caused me considerable surprise. With the help of Stal Aanderaa [22], I was reduced to showing that multiplication requires $\Omega(n \log n / (\log \log n)^2)$ steps using an "on-line" Turing machine.⁴ I also pointed out in my thesis that Toom's method can be adapted to multitape Turing machines in order to multiply in $O(n^{1+\epsilon})$ steps, something that I am sure came as no surprise to Toom.

The currently fastest asymptotic running time on a multitape Turing machine for number multiplication is $O(n \log \log n)$, and was devised by Schönhage and Strassen [70] (1971) using the fast Fourier Transform. However, Schönhage [69] recently showed by a complicated argument that his storage modification machines (see Section 2) can multiply in time $O(n)$ (linear time!). We are forced to conclude that either multiplication is easier than we thought or that Schönhage's machines cheat.

(3) **Matrix multiplication.** The obvious method requires $n^2(2n-1)$ arithmetic operations to multiply two $n \times n$ matrices, and attempts were made to prove the method optimal in the 1950's and 1960's. There was surprise when Strassen [81] (1969) published his method requiring only $4.7n^{2.37}$ operations. Considerable work has been devoted to reducing the exponent of 2.81, and currently the best time known is $O(n^{2.37})$ operations, due to Coppersmith and Winograd [24]. There is still plenty of room for progress, since the best known lower bound is $2n^2 \cdot 1$ (see [13]).

(4) **Maximum matchings in general undirected graphs.** This was perhaps the first problem explicitly shown to be in P whose membership in P requires a difficult algorithm. Edmonds' influential paper [27] gave the result and discussed the notion of a polynomial time algorithm (see Section 2). He also pointed out that the simple notion of augmenting path, which suffices for the bipartite case, does not work for general undirected graphs.

(5) **Recognition of prime numbers.** The major question here is whether this problem is in P . In other words, is there an algorithm that always tells us

whether an arbitrary n -digit input integer is prime, and halts in a number of steps bounded by a fixed polynomial in n ? Gary Miller [53] (1976) showed that there is such an algorithm, but its validity depends on the extended Riemann hypothesis. Solovay and Strassen [77] devised a fast Monte Carlo algorithm (see Section 5) for prime recognition, but if the input number is composite there is a small chance the algorithm will mistakenly say it is prime. The best provable deterministic algorithm known is due to Adleman, Pomerance, and Rumely [2] and runs in time $n^{O(\log \log n)}$, which is slightly worse than polynomial. A variation of this due to H. Cohen and H. W. Lenstra Jr. [17] can routinely handle numbers up to 100 decimal digits in approximately 45 seconds.

Recently three important problems have been shown to be in the class P . The first is linear programming, shown by Khachian [43] in 1979 (see [55] for an exposition). The second is determining whether two graphs of degree at most d are isomorphic, shown by Luks [50] in 1980. (The algorithm is polynomial in the number of vertices for fixed d , but exponential in d .) The third is factoring polynomials with rational coefficients. This was shown for polynomials in one variable by Lenstra, Lenstra, and Lovasz [48] in 1982. It can be generalized to polynomials in any fixed number of variables as shown by Kaltfofen's result [39], [40].

4. LOWER BOUNDS

The real challenge in complexity theory, and the problem that sets the theory apart from the analysis of algorithms, is proving lower bounds on the complexity of specific problems. There is something very satisfying in proving that a yes-no problem cannot be solved in n , or n^2 , or 2^n steps, no matter what algorithm is used. There have been some important successes in proving lower bounds, but the open questions are even more important and somewhat frustrating.

All important lower bounds on computation time or space are based on "diagonal arguments." Diagonal arguments were used by Turing and his contemporaries to prove certain problems are not algorithmically solvable. They were also used prior to 1960 to define hierarchies of computable 0-1 functions.⁵ In 1960, Rabin [60] proved that for any reasonable complexity measure, such as computation time or space (memory), sufficiently increasing the allowed time or space etc. always allows more 0-1 functions to be computed. About the same time, Ritchie in his thesis [65] defined a specific hierarchy of functions (which he showed is nontrivial for 0-1 functions) in terms of the amount of space allowed. A little later Rabin's result was amplified in detail for time on multitape Turing machines by Hartmanis and Stearns [37], and for space by Stearns, Hartmanis, and Lewis [78].

4.1 Natural Decidable Problems Proved Infeasible

The hierarchy results mentioned above gave lower bounds on the time and space needed to compute specific functions, but all such functions seemed to be "contrived." For example, it is easy to see that the function $f(x,y)$ which gives the first digit of the output of machine x on input y after $(|x|+|y|)^2$ steps cannot be computed in time $(|x|+|y|)^2$. It was not until 1972, when Albert Meyer and Larry Stockmeyer [52] proved that the equivalence problem for regular expressions with squaring requires exponential space and, therefore, exponential time, that a nontrivial lower bound for general models of computation on a "natural" problem was found (natural in

⁴This lower bound has been slightly improved. See [56] and [64].

⁵See, for example, Grzegorczyk [35].

the sense of being interesting, and not about computing machines]. Shortly after that Meyer [51] found a very strong lower bound on the time required to determine the truth of formulas in a certain formal decidable theory called WSIS (weak monadic second-order theory of successor). He proved that any computer whose running time was bounded by a fixed number of exponentials (2^2 , 2^{2^2} , $2^{2^{2^2}}$, etc.) could not correctly decide WSIS. Meyer's Ph.D. student, Stockmeyer went on to calculate [79] that any Boolean circuit (think computer) that correctly decides the truth of an arbitrary WSIS formula of length 616 symbols must have more than 10^{121} gates. The number 10^{121} was chosen to be the number of protons that could fit in the known universe. This is a very convincing infeasibility proof!

Since Meyer and Stockmeyer there have been a large number of lower bounds on the complexity of decidable formal theories (see [29] and [80] for summaries). One of the most interesting is a doubly exponential time lower bound on the time required to decide Presburger arithmetic (the theory of the natural numbers under addition) by Fischer and Rabin [30]. This is not far from the best known time upper bound for this theory, which is triply exponential [54]. The best space upper bound is doubly exponential [29].

Despite the above successes, the record for proving lower bounds on problems of smaller complexity is appalling. In fact, there is no nonlinear time lower bound known on a general purpose computation model for any natural problem in NP (See section 4.4), in particular, for any of the 300 problems listed in [31]. Of course, one can prove by diagonal arguments the existence of problems in NP requiring time n^k for any fixed k . In the case of space lower bounds, however, we do not even know how to prove the existence of NP problems not solvable in space $O(\log n)$ on an off-line Turing machine (see Section 4.3). This is despite the fact that the best known space upper bounds in many natural cases are essentially linear in n .

4.2 Structured Lower Bounds

Although we have had little success in proving interesting lower bounds for concrete problems on general computer models, we do have interesting results for "structured" models. The term "structured" was introduced by Borodin [9] to refer to computers restricted to certain operations appropriate to the problem at hand. A simple example of this is the problem of sorting n numbers. One can prove (see [44]) without much difficulty that this requires at least $n \log n$ comparisons, provided that the only operation the computer is allowed to do with the inputs is to compare them in pairs. This lower bound says nothing about Turing machines or Boolean circuits, but it has been extended to unit cost random access machines, provided division is disallowed.

A second and very elegant structured lower bound, due to Strassen [82] (1973), states that polynomial interpolation, that is, finding the coefficients of the polynomial of degree $n-1$ that passes through n given points, requires $\Omega(n \log n)$ multiplications, provided only arithmetic operations are allowed. Part of the interest here is that Strassen's original proof depends on Bezout's theorem, a deep result in algebraic geometry. Very recently, Baur and Strassen [83] have extended the lower bound to show that even the middle coefficient of the interpolating polynomial through n points requires $\Omega(n \log n)$ multiplications to compute.

Part of the appeal of all of these structured results is that the lower bounds are close to the best known upper bounds,⁶ and the best known algorithms can be implemented on the

structured models to which the lower bounds apply.

4.3 Time-Space Product Lower Bounds

Another way around the impasse of proving time and lower bounds is to prove time lower bounds under the assumption of small space. Cobham [16] proved the first result in 1966, when he showed that the time-space product for recognizing n -digit perfect squares on an "off-line" machine must be $\Omega(n^2)$. (The same is true of n -symbol alphabets.) Here the input is written on a two-way read input tape, and the space used is by definition the number of squares scanned by the work tapes available to the T_2 machine. Thus, if, for example, the space is restricted to $O(\log n)$ (which is more than sufficient), then the time is $\Omega(n/\log n)$ steps.

The weakness in Cobham's result is that although the off-line Turing machine model is a reasonable one for the computation time or space separately, it is too restrictive when time and space are considered together. For example, the palindromes can obviously be recognized in $2n$ steps in constant space if two heads are allowed to scan the input simultaneously. Borodin and I [110] partially rectified this weakness when we proved that sorting n integers in the range one to n^2 requires a time-space product of $\Omega(n^2)$. The proof applies to any "general sequential machine," which includes off-line Turing machines with many input heads and even random access to the input tape. It is unfortunate that it is crucial to our proof that sorting requires many output heads, and it remains an interesting open question whether a lower bound can be made to apply to a set recognition problem, such as recognizing whether all n input numbers are distinct. (Our lower bound on sorting has recently been slightly improved in [64].)

4.4 NP-Completeness

The theory of NP-completeness is surely the most significant development in computational complexity. I will not cover it here because it is now well known and is the subject of many textbooks. In particular, the book by Garey and Johnson [29] is an excellent place to read about it.

The class NP consists of all sets recognizable in polynomial time by a nondeterministic Turing machine. As far as I am concerned, the first time a mathematically equivalent class was defined was by James Bennett in his 1962 Ph.D. thesis [4]. Bennett used the name "extended positive rudimentary relation" for his class, and his definition used logical quantifiers instead of the standard ones used for computing machines. I read this part of his thesis and I think his class could be characterized as the now familiar definition of NP. I used the term Σ^P (after Cobham's class Σ) in my 1971 paper [18], and Karp gave the now accepted name for the class in his 1972 paper [42]. Meanwhile, quite independently of the formal development, Edmonds, back in 1965 talked informally about problems with a "good characterization," a notion essentially equivalent to NP.

In 1971 [18], I introduced the notion of NP-complete and proved 3-satisfiability and the subgraph problem were NP-complete. A year later, Karp [42] proved 21 problems were NP-complete, thus forcefully demonstrating the importance of the subject. Independently of this and slightly later, Leonid Levin [49], in the Soviet Union (now at Boston University), defined a similar (and stronger) notion and proved six problems were complete in his sense. The informal notion of "search problem" was standard in the Soviet literature,

⁶ See Borodin and Munro [12] for upper bounds for interpolation.

Levin called his problems "universal search problems."

The class NP includes an enormous number of practical problems that occur in business and industry (see [31]). A proof that an NP problem is NP-complete is a proof that the problem is not in P (does not have a deterministic polynomial time algorithm) unless every NP problem is in P. Since the latter condition would revolutionize computer science, the practical effect of NP-completeness is a lower bound. This is why I have included this subject in the section on lower bounds.

4.5 #P-Completeness

The notion of NP-completeness applies to sets, and a proof that a set is NP-complete is usually interpreted as a proof that it is intractable. There are, however, a large number of apparently intractable functions for which no NP-completeness proof seems to be relevant. Leslie Valiant [86, 87] defined the notion of #P-completeness to help remedy this situation. Proving that a function is #P-complete shows that it is apparently intractable to compute in the same way that proving a set is NP-complete shows that it is apparently intractable to recognize; namely, if a #P-complete function is computable in polynomial time, then $P = NP$.

Valiant gave many examples of #P-complete functions, but probably the most interesting one is the permanent of an integer matrix. The permanent has a definition formally similar to the determinant, but whereas the determinant is easy to compute by Gaussian elimination, the many attempts over the past hundred odd years to find a feasible way to compute the permanent have all failed. Valiant gave the first convincing reason for this failure when he proved the permanent #P-complete.

5. PROBABILISTIC ALGORITHMS

The use of random numbers to simulate or approximate random processes is very natural and is well established in computing practice. However, the idea that random inputs might be very useful in solving deterministic combinatorial problems has been much slower in penetrating the computer science community. Here I will restrict attention to probabilistic (coin tossing) polynomial time algorithms that "solve" (in a reasonable sense) a problem for which no deterministic polynomial time algorithm is known.

The first such algorithm seems to be the one by Berlekamp [5] in 1970, for factoring a polynomial f over the field $GF(p)$ of p elements. Berlekamp's algorithm runs in time polynomial in the degree of f and $\log p$, and with probability at least one-half it finds a correct prime factorization of f ; otherwise it ends in failure. Since the algorithm can be repeated any number of times and the failure events are all independent, the algorithm in practice always factors in a feasible amount of time.

A more dramatic example is the algorithm for prime recognition due to Solovay and Strassen [77] (submitted in 1974). This algorithm runs in time polynomial in the length of the input m , and outputs either "prime" or "composite." If m is in fact prime, then the output is certainly "prime," but if m is composite, then with probability at most one-half the answer may also be "prime." The algorithm may be repeated any number of times on an input m with independent results. Thus if the answer is ever "composite," the user knows m is composite; if the answer is consistently "prime" after, say, 100 runs, then the user has good evidence that m is prime, since any fixed composite m would give such results with tiny probability (less than 2^{-100}).

Rabin [61] developed a different probabilistic algorithm

with properties similar to the one above, and found it to be very fast on computer trials. The number $2^{39} - 593$ was identified as (probably) prime within a few minutes.

One interesting application of probabilistic prime testers was proposed by Rivest, Shamir, and Adleman [67a] in their landmark paper on public key cryptosystems in 1978. Their system requires the generation of large (100 digit) random primes. They proposed testing random 100 digit numbers using the Solovay-Strassen method until one was found that was probably prime in the sense outlined above. Actually with the new high powered deterministic prime tester of Cohen and Lenstra [17] mentioned in Section 3, once a random 100 digit "probably prime" number was found it could be tested for certain in about 45 seconds, if it is important to know for certain.

The class of sets with polynomial time probabilistic recognition algorithms in the sense of Solovay and Strassen is known as R (or sometimes RP) in the literature. Thus a set is in R if and only if it has a probabilistic recognition algorithm that always halts in polynomial time and never makes a mistake for inputs not in R , and for each input in R it outputs the right answer for each run with probability at least one-half. Hence the set of composite numbers is in R , and in general $P \subseteq R \subseteq NP$. There are other interesting examples of sets in R not known to be in P . For example, Schwartz [71] shows that the set of nonsingular matrices whose entries are polynomials in many variables is in R . The algorithm evaluates the polynomials at random small integer values and computes the determinant of the result. (The determinant apparently cannot feasibly be computed directly because the polynomials computed would have exponentially many terms in general.)

It is an intriguing open question whether $R = P$. It is tempting to conjecture yes on the philosophical grounds that random coin tosses should not be of much use when the answer being sought is a well defined yes or no. A related question is whether a probabilistic algorithm (showing a problem is in R) is for all practical purposes as good as a deterministic algorithm. After all, the probabilistic algorithms can be run using the pseudorandom number generations available on most computers, and an error probability of 2^{-100} is negligible. The catch is that pseudorandom number generators do not produce truly random numbers, and nobody knows how well they will work for a given probabilistic algorithm. In fact, experience shows they seem to work well. But if they always work well, then it follows that $R = P$, because pseudorandom numbers are generated deterministically so true randomness would not help after all. Another possibility is to use a physical process such as thermal noise to generate random numbers. But it is an open question in the philosophy of science how truly random nature can be.

Let me close this section by mentioning an interesting theorem of Adleman [1] on the class R . It is easy to see [57b] that if a set is in P , then for each n there is a Boolean circuit of size bounded by a fixed polynomial in n which determines whether an arbitrary string of length n is in the set. What Adleman proved is that the same is true for the class R . Thus, for example, for each n there is a small "computer circuit" that correctly and rapidly tests whether n digit numbers are prime. The catch is that the circuits are not uniform in n , and in fact for the case of 100 digits it may not be feasible to figure out how to build the circuit.⁷

6. SYNCHRONOUS PARALLEL COMPUTATION

With the advent of VLSI technology in which one or more processors can be placed on a quarter-inch chip, it is natural

⁷ For more theory on probabilistic computation, see Gill [32].

to think of a future computer composed of many thousands of such processors working together in parallel to solve a single problem. Although no very large general purpose machine of this kind has been built yet, there are such projects under way (see Schwartz [72]). This motivates the recent development of a very pleasing branch of computation complexity: the theory of large scale synchronous parallel computation, in which the number of processors is a resource bounded by a parameter $H(n)$ (H is for hardware) in the same way that space is bounded by a parameter $S(n)$ in sequential complexity theory. Typically $H(n)$ is a fixed polynomial in n .

Quite a number of parallel computation models have been proposed (see [21] for a review), just as there are many competing sequential models (see Section 2). There are two main contenders, however. The first is the class of shared memory models in which a large number of processors communicate via a random access memory that they hold in common. Many parallel algorithms have been published for such models, since real parallel machines may well be like this when they are built. However, for the mathematical theory these models are not very satisfactory because too much of their detailed specification is arbitrary: How are read and write conflicts in the common memory resolved? What basic operations are allowed for each processor? Should one charge $H(n)$ time units to access common memory?

Hence I prefer the cleaner model discussed by Borodin [8] (1977), in which a parallel computer is a uniform family $\langle B_n \rangle$ of acyclic Boolean circuits, such that B_n has n inputs (and hence takes care of those input strings of length n). Then $H(n)$ (the amount of hardware) is simply the number of gates in B_n , and $T(n)$ (the parallel computation time) is the depth of the circuit B_n (i.e., length of the longest path from an input to an output). This model has the practical justification that presumably all real machines (including shared memory machines) are built from Boolean circuits. Furthermore, the minimum Boolean size and depth needed to compute a function is a natural mathematical problem and was considered well before the theory of parallel computation was around.

Fortunately for the theory, the minimum values of hardware $H(n)$ and parallel time $T(n)$ are not wildly different for the various competing parallel computer models. In particular, there is an interesting general fact true for all the models, first proved for a particular model by Pratt and Stockmeyer [58] in 1974 and called the "parallel computation thesis" in [33]: namely, a problem can be solved in time polynomial in $T(n)$ by a parallel machine (with unlimited hardware) if and only if it can be solved in space polynomial in $T(n)$ by a sequential machine (with unlimited time).

A basic question in parallel computation is: Which problems can be solved substantially faster using many processors rather than one processor? Nicholas Pippenger [57a] formalized this question by defining the class (now called NC, for "Nick's class") of problems solvable ultra fast [time $T(n) = (\log n)^{\alpha}$] on a parallel computer with a feasible [$H(n) = n^{\alpha}$] amount of hardware. Fortunately, the class NC remains the same, independent of the particular parallel computer model chosen, and it is easy to see that NC is a subset of the class FP of functions computable sequentially in polynomial time. Our informal question can then be formalized as follows: Which problems in FP are also in NC?

It is conceivable (though unlikely) that $NC = FP$, since to prove $NC \neq FP$ would require a breakthrough in complexity theory (see the end of Section 4.1). Since we do not know how to prove a function f in FP is not in NC, the next best thing is to prove that f is log space complete for FP. This is the analog of proving a problem is NP-complete, and has the

practical effect of discouraging efforts for finding super fast parallel algorithms for f . This is because if f is log space-complete for FP and f is in NC, then $FP = NC$, which would be a big surprise.

Quite a bit of progress has been made in classifying problems in FP as to whether they are in NC or log space-complete for FP (of course, they may be neither). The first example of a problem complete for P was presented in 1973 by me in [20], although I did not state the result as a completeness result. Shortly after that Jones and Laaser [38] defined this notion of completeness and gave about five examples, including the emptiness problem for context-free grammars. Probably the simplest problem proved complete for FP is the so-called circuit value problem [47]: given a Boolean circuit together with values for its inputs, find the value of the output. The example most interesting to me, due to Goldschlager, Shaw, and Staples [34], is finding the (parity of) the maximum flow through a given network with (large) positive integer capacities on its edges. The interest comes from the subtlety of the completeness proof. Finally, I should mention that linear programming is complete for FP. In this case the difficult part is showing that the problem is in P (see [43]), after which the completeness proof [26] is straightforward.

Among the problems known to be in NC are the four arithmetic operations ($+$, $-$, \cdot , $/$) on binary numbers, sorting, graph connectivity, matrix operations (multiplication, inverse, determinant, rank), polynomial greatest common divisors, context-free languages, and finding a minimum spanning forest in a graph (see [11], [21], [63], [67b]). The size of a maximum matching for a given graph is known [11] to be in "random" NC (NC in which coin tosses are allowed), although it is an interesting open question of whether finding an actual maximum matching is even in random NC. Results in [89] and [67b] provide general methods for showing problems are in NC.

The most interesting problem in FP not known either to be complete for FP or in (random) NC is finding the greatest common divisor of two integers. There are many other interesting problems that have yet to be classified, including finding a maximum matching or a maximal clique in a graph (see [88]).

7. THE FUTURE

Let me say again that the field of computational complexity is large and this overview is brief. There are large parts of the subject that I have left out altogether or barely touched on. My apologies to the researchers in those parts.

One relatively new and exciting part, called "computational information theory," by Yao [92], builds on Shannon's classical information theory by considering information that can be accessed through a feasible computation. This subject was sparked largely by the papers by Diffie and Hellman [25] and Rivest, Shamir, and Adleman [67a] on public key cryptosystems, although its computational roots go back to Kolmogoroff [45] and Chaitin [14a], [14b], who first gave meaning to the notion of a single finite sequence being "random," by using the theory of computation. An interesting idea in this theory, considered by Shamir [73] and Blum and Micali [7], concerns generating pseudorandom sequences in which future bits are provably hard to predict in terms of past bits. Yao [92] proves that the existence of such sequences would have positive implications about the deterministic complexity of the probabilistic class R (see Section 5). In fact, computational information theory promises to shed light on the role of randomness in computation.

In addition to computational information theory we can

expect interesting new results on probabilistic algorithms, parallel computation, and (with any luck) lower bounds. Concerning lower bounds, the one breakthrough for which I see some hope in the near future is showing that not every problem in P is solvable in space $O(\log n)$, and perhaps also $P \neq NC$. In any case, the field of computational complexity remains very vigorous, and I look forward to seeing what the future will bring.

Acknowledgments. I am grateful to my complexity colleagues at Toronto for many helpful comments and suggestions, especially Allan Borodin, Joachim von zur Gathen, Silvio Micali, and Charles Rackoff.

REFERENCES

1. Adleman, L. Two theorems on random polynomial time. Proc. 19th IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1978), 75–83.
2. Adleman, L., Pomerance, C., and Rumley, R. S. On distinguishing prime numbers from composite numbers. *Annals of Math* 117, (January 1983), 173–206.
3. Aho, A. V., Hopcroft, J. E. and Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
4. Bennett, J. H. On Spectra. Doctoral dissertation, Department of Mathematics, Princeton University, 1962.
5. Berlekamp, E. R. Factoring polynomials over large finite fields. *Math. Comp.* 24 (1970), 713–735.
6. Blum, M. A machine independent theory of the complexity of recursive functions. *JACM* 14, 2 (April, 1967), 322–336.
7. Blum, M., and Micali, S. How to generate cryptographically strong sequences of pseudo random bits. Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1982), 112–117.
8. Borodin, A. On relating time and space to size and depth. *SIAM J. Comp.* 6 (1977), 733–744.
9. Borodin, A. Structured vs. general models in computational complexity. In *Logic and Algorithmic Monographie no. 30 de L'Enseignement Mathématique Université de Genève*, 1982.
10. Borodin, A. and Cook S. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.* 11 (1982), 287–297.
11. Borodin, A., von zur Gathen, J., and Hopcroft, J. Fast parallel matrix and GCD computations. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1982), 63–71.
12. Borodin, A. and Munro, I. *The Computational Complexity of Algebraic and Numeric Problems*. Elsevier, New York, 1975.
13. Brockett, R. W. and Dobkin, D. On the optimal evaluation of a set of bilinear forms. *Linear Algebra and its Applications* 19 (1978), 207–235.
- 14a. Chaitin, G. J. On the length of programs for computing finite binary sequences. *JACM* 13, 4 (October 1966), 547–569; *JACM* 16, 1 (January 1969), 145–159.
- 14b. Chaitin, G. J. A theory of program size formally identical to informational theory. *JACM* 22, 3 (July 1975), 329–340.
15. Cobham, A. The intrinsic computational difficulty of functions. Proc. 1964 International Congress for Logic, Methodology, and Philosophy of Sciences. Y. Bar-Hillel, Ed., North Holland, Amsterdam, 1965, 24–30.
16. Cobham, A. The recognition problem for the set of perfect squares. IEEE Conference Record Seventh SWAT. (1966), 78–87.
17. Cohen, H. and Lenstra, H. W. Jr. Primarily testing and Jacobi sums. Report 82–18, University of Amsterdam, Dept. of Math., 1982.
18. Cook, S. A. The complexity of theorem proving procedures. Proc. 3rd ACM Symp. on Theory of Computing. Shaker Heights, Ohio, (May 3–5, 1971), 151–158.
19. Cook, S. A. Linear time simulation of deterministic two-way pushdown automata. Proc. IFIP Congress 71, (Theoretical Foundations). North Holland, Amsterdam, 1972, 75–80.
20. Cook, S. A. An observation on time-storage tradeoff. *JCSS* 9 (1974), 308–316. Originally in Proc. 5th ACM Symp. on Theory of Computing. Austin TX. (April 30–May 2 1973) 29–33.
21. Cook, S. A. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathématique XXVII* (1981), 99–124.
22. Cook, S. A. and Aanderaa, S. O. On the minimum computation time of functions. *Trans. AMS* 142 (1969), 291–314.
23. Cooley, J. M. and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* 19 (1965), 297–301.
24. Coppersmith, D. and Winograd, S. On the asymptotic complexity of matrix multiplication. *SIAM J. Comp.* 11 (1982), 472–492.
25. Diffie, W. and Hellman, M. E. New directions in cryptography. *IEEE Trans. on Inform. Theory* IT-22 6 (1976), 644–654.
26. Dobkin, D., Lipton, R. J. and Reiss, S. Linear programming is log-space hard for P . *Inf. Processing Letters* 8 (1979), 96–97.
27. Edmonds, J. Paths, trees, flowers. *Canad. J. Math.* 17 (1965), 449–467.
28. Edmonds, J. Minimum partition of a matroid into independent subsets. *J. Res. Natl. Bur. Standards Sect. B*, 69 (1965), 67–72.
29. Ferrante, J. and Rackoff, C. W. *The Computational Complexity of Logical Theories*. Lecture Notes in Mathematics, #718, Springer Verlag, New York, 1979.
30. Fischer, M. J. and Rabin, M. O. Super-exponential complexity of Presburger arithmetic. In *Complexity of Computation*. SIAM–AMS Proc. 7, R. Karp, Ed., 1974, 27–42.
31. Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.
32. Gill, I. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.* 6 (1977), 675–695.
33. Goldschlager, L. M. *Synchronous Parallel Computation*. Doctoral dissertation, Dept. of Computer Science, Univ. of Toronto, 1977. See also *JACM* 29, 4, (October 1982), 1073–1086.
34. Goldschlager, L. M., Shaw, R. A. and Staples, J. The maximum flow problem is log space complete for P . *Theoretical Computer Science* 21 (1982), 105–111.
35. Grzegorczyk, A. Some classes of recursive functions. *Rozprawy Matematyczne*, 1953.
36. Hartmann, J. Observations about the development of theoretical computer science. *Annals Hist. Comput.* 3, 1 (Jan. 1981), 42–51.
37. Hartmann, J. and Stearns, R. E. On the computational complexity of algorithms. *Trans. AMS* 117 (1965), 285–306.
38. Jones, N. D. and Laaser, W. T. Complete problems for deterministic polynomial time. *Theoretical Computer Science* 3 (1977), 105–117.
39. Kaltofen, E. A polynomial reduction from multivariate to bivariate integer polynomial factorization. Proc. 14th ACM Symp. in Theory Comp. San Francisco, CA, (May 5–7 1982), 261–266.
40. Kaltofen, E. A polynomial-time reduction from bivariate to univariate integral polynomial factorization. Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles 1982, 57–64.
41. Karatsuba, A. and Ofman, Yu. Multiplication of multidigit numbers on automata. *Doklady Akad. Nauk* 145, 2 (1962), 293–294. Translated in Soviet Phys. *Doklady* 7, 7 (1963), 595–596.
42. Karp, R. M. Reducibility among combinatorial problems. In: *Complexity of Computer Computations*. R. E. Miller and J. W. Thatcher, Eds., Plenum Press, New York, 1972, 85–104.
43. Khachian, L. G. A polynomial time algorithm for linear programming. *Doklady Akad. Nauk SSSR* 244, 5 (1979), 1093–96. Translated in Soviet Math. *Doklady*, 20, 191–194.
44. Knuth, D. E. *The Art of Computer Programming*, vol. 3 *Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.
45. Kolmogorov, A. N. Three approaches to the concept of the amount of information. *Probl. Pered. Inf. (Probl. of Inf. Transm.)* 1 (1965).
46. Kolmogorov, A. N. and Uspenski, V. A. On the definition of an algorithm. *Uspehi Mat. Nauk.* 13 (1958), 3–28; *AMS Transl.* 2nd ser. 29 (1963), 217–245.
47. Ladner, R. E. The circuit value problem is log space complete for P . *SIGACT News* 7, 1 (1975), 18–20.
48. Lenstra, A. K., Lenstra, H. W. and Lovasz, L. Factoring polynomials with rational coefficients. Report 82–05, University of Amsterdam, Dept. of Math., 1982.
49. Levin, L. A. Universal search problems. *Problemy Peredaci Informacii* 9 (1973), 115–116. Translated in *Problems of Information Transmission* 9, 265–266.
50. Luks, E. M. Isomorphism of graphs of bounded valence can be tested in polynomial time. Proc. 21st IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1980), 42–49.
51. Meyer, A. R. Weak monadic second-order theory of successor is not elementary-recursive. *Lecture Notes in Mathematics*, #453, Springer Verlag, New York, 1975, 132–154.
52. Meyer, A. R. and Stockmeyer, L. J. The equivalence problem for regular expressions with squaring requires exponential space. Proc. 13th IEEE Symp. on Switching and Automata Theory. (1972), 125–129.
53. Miller, G. I. Riemann's Hypothesis and tests for primality. *J. Comput. System Sci.* 13 (1976), 300–317.
54. Oppen, D. C. A $2^{2^{2^P}}$ upper bound on the complexity of Presburger arithmetic. *J. Comput. Syst. Sci.* 16 (1978), 323–332.
55. Papadimitriou, C. H. and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.
56. Paterson, M. S., Fischer, M. J. and Meyer, A. R. An improved overlap argument for on-line multiplication. *SIAM–AMS Proc. 7*, Amer. Math. Soc., Providence, 1974, 97–111.
- 57a. Pippenger, N. On simultaneous resource bounds (preliminary version). Proc. 20th IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1979), 307–311.
- 57b. Pippenger, N. J. and Fischer, M. J. Relations among complexity measures. *JACM* 26, 2 (April 1979), 361–381.

58. Pratt, V. R. and Stockmeyer, L. J. A characterization of the power of vector machines. *J. Comput. System Sci.* 12 (1976), 198-221. Originally in Proc. 5th ACM Symp. on Theory of Computing, Seattle, WA, (April 30-May 2, 1974), 122-134.
59. Rabin, M. O. Speed of computation and classification of recursive sets. *Third Convention Sci. Soc., Israel*, 1959, 1-2.
60. Rabin, M. O. Degree of difficulty of computing a function and a partial ordering of recursive sets. Tech. Rep. No. 1, O.N.R., Jerusalem, 1960.
61. Rabin, M. O. Probabilistic algorithms. In *Algorithms and Complexity, New Directions and Recent Trends*. J. F. Traub, Ed., Academic Press, New York, 1976, 21-39.
62. Rabin, M. O. Complexity of Computations. *Comm. ACM* 20, 9 (September 1977), 625-633.
63. Reif, J. H. Symmetric Complementation. Proc. 14th ACM Symp. on Theory of Computing, San Francisco, CA, (May 5-7, 1982), 201-214.
64. Reisch, S. and Schnitger, G. Three applications of Kolmogorov complexity. Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles, 1982, 45-52.
65. Ritchie, R. W. *Glosses of Recursive Functions of Predictable Complexity*. Doctoral Dissertation, Princeton University, 1960.
66. Ritchie, R. W. Classes of predictably computable functions. *Trans. AMS* 106 (1963), 139-173.
- 67a. Rivest, R. L., Shamir, A. and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM* 21, 2 (February 1978), 120-126.
- 67b. Ruzzo, W. L. On uniform circuit complexity. *J. Comput. System Sci.* 22 (1981), 365-383.
- 68a. Savage, J. E. *The Complexity of Computing*. Wiley, New York, 1976.
- 68b. Schnorr, C. P. The network complexity and the Turing machine complexity of finite functions. *Acta Informatica* 7 (1976), 95-107.
69. Schönhage, A. Storage modification machines. *SIAM J. Comp.* 9 (1980), 490-508.
70. Schönhage, A. and Strassen, V. Schnelle Multiplikation grosser Zahlen. *Computing* 7 (1971), 281-292.
71. Schwartz, J. T. Probabilistic algorithms for verification of polynomial identities. *JACM* 27, 4 (October 1980), 701-717.
72. Schwartz, J. T. Ultracomputers. *ACM Trans. on Prog. Languages and Systems* 2, 4 (October 1980), 484-521.
73. Shamir, A. On the generation of cryptographically strong pseudo random sequences. 8th Int. Colloquium on Automata, Languages, and Programming (July 1981), Lecture Notes in Computer Science No. 115, Springer Verlag, New York, 544-550.
74. Shannon, C. E. The synthesis of two terminal switching circuits. *BSTJ* 28 (1949), 59-98.
75. Smale, S. On the average speed of the simplex method of linear programming. Preprint, 1982.
76. Smale, S. The problem of the average speed of the simplex method. Preprint, 1982.
77. Solovay, R. and Strassen, V. A fast monte-carlo test for primality. *SIAM J. Comput.* 6 (1977), 84-85.
78. Stearns, R. E., Hartmanis, J. and Lewis P. M. II. Hierarchies of memory limited computations. 6th IEEE Symp. on Switching Circuit Theory and Logical Design, (1966), 179-190.
79. Stockmeyer, L. J. The complexity of decision problems in automata theory and logic. Doctoral Thesis, Dept. of Electrical Eng., MIT, Cambridge, MA., 1974; Report TR-133, MIT Laboratory for Computer Science.
80. Stockmeyer, L. J. Classifying the computational complexity of problems. Research Report RC 7606 (1979), Math. Sciences Dept., IBM T. J. Watson Research Center, Yorktown Heights, N.Y.
81. Strassen, V. Gaussian elimination is not optimal. *Num. Math.* 13 (1969), 354-356.
82. Strassen, V. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. *Numer. Math.* 20 (1973), 238-251.
83. Baur, W. and Strassen, V. The complexity of partial derivatives. Preprint, 1982.
84. Toom, A. I. The complexity of a scheme of functional elements realizing the multiplication of integers. *Doklady Akad. Nauk. SSSR* 150 3, (1963) 496-498. Translated in *Soviet Math. Doklady* 3 (1963), 714-716.
85. Turing, A. M. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc.* ser. 2, 42 (1936-7), 230-265. A correction, *ibid.* 43 (1937), 544-546.
86. Valiant, L. G. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8 (1979), 410-421.
87. Valiant, L. G. The complexity of computing the permanent. *Theoretical Computer Science* 8 (1979), 189-202.
88. Valiant, L. G. Parallel Computation. Proc. 7th IBM Japan Symp. Academic 6 Scientific Programs, IBM Japan, Tokyo (1982).
89. Valiant, L. G., Skyum, S., Berkowitz, S. and Rackoff, C. Fast parallel computation on polynomials using few processors. Preprint (Preliminary version in Springer Lecture Notes in Computer Science, 118 (1981), 132-139.
90. von Neumann, J. A certain zero-sum two-person game equivalent to the optimal assignment problem. Contributions to the Theory of Games II. H. W. Kuhn and A. W. Tucker, Eds. Princeton Univ. Press, Princeton, NJ, 1953.
91. Yamada, H. Real time computation and recursive functions not real-time computable. *IRE Transactions on Electronic Computers*. EC-11 (1962), 753-760.
92. Yao, A. C. Theory and applications of trapdoor functions [Extended abstract]. Proc. 23rd IEEE Symp. on Foundations of Computer Science. IEEE Computer Society, Los Angeles (1982), 80-91.

CR Categories and Subject Descriptors: F. O. [General]

General Term: Theory

Additional Key Words and Phrases: computational complexity

ACM Algorithms

Collected Algorithms from ACM (CALGO) now includes quarterly issues of complete algorithm listings on microfiche as part of the regular CALGO supplement service.

The ACM Algorithms Distribution Service now offers microfiche containing complete listings of ACM algorithms, and also offers compilations of algorithms on tape as a substitute for tapes containing single algorithms. The fiche and tape compilations are available by quarter and by year. Tape compilations covering five years will also be available.

To subscribe to CALGO, request an order form and a free ACM Publications Catalog from the ACM Subscription Department, Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036. To order from the ACM Algorithms Distributions Service, refer to the order form that appears in every issue of **ACM Transactions on Mathematical Software**.



1975 ACM Turing Award Lecture

The 1975 ACM Turing Award was presented jointly to Allen Newell and Herbert A. Simon at the ACM Annual Conference in Minneapolis, October 20. In introducing the recipients, Bernard A. Galler, Chairman of the Turing Award Committee, read the following citation:

"It is a privilege to be able to present the ACM Turing Award to two friends of long standing, Professors Allen Newell and Herbert A. Simon, both of Carnegie-Mellon University.

"In joint scientific efforts extending over twenty years, initially in collaboration with J.C. Shaw at the RAND Corporation, and subsequently with numerous faculty and student colleagues at Carnegie-Mellon University, they have made basic contributions to artificial intelligence, the psychology of human cognition, and list processing.

"In artificial intelligence, they contributed to the establishment of the field as an area of scientific endeavor, to the development of heuristic programming generally, and of heuristic search, means-ends analysis, and methods of induction, in particular; providing

demonstrations of the sufficiency of these mechanisms to solve interesting problems.

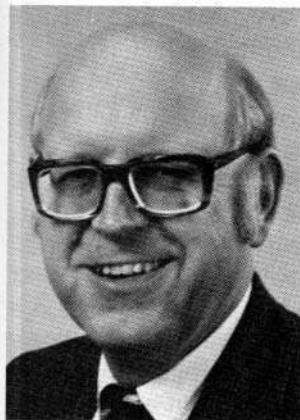
"In psychology, they were principal instigators of the idea that human cognition can be described in terms of a symbol system, and they have developed detailed theories for human problem solving, verbal learning and inductive behavior in a number of task domains, using computer programs embodying these theories to simulate the human behavior.

"They were apparently the inventors of list processing, and have been major contributors to both software technology and the development of the concept of the computer as a system of manipulating symbolic structures and not just as a processor of numerical data.

"It is an honor for Professors Newell and Simon to be given this award, but it is also an honor for ACM to be able to add their names to our list of recipients, since by their presence, they will add to the prestige and importance of the ACM Turing Award."

Computer Science as Empirical Inquiry: Symbols and Search

Allen Newell and Herbert A. Simon



Key Words and Phrases: symbols, search, science, computer science, empirical, Turing, artificial intelligence, intelligence, list processing, cognition, heuristics, problem solving.

CR Categories: I.0, 2.1, 3.3, 3.6, 5.7.

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication,

Computer science is the study of the phenomena surrounding computers. The founders of this society understood this very well when they called themselves the Association for Computing Machinery. The machine—not just the hardware, but the programmed, living machine—is the organism we study.

This is the tenth Turing Lecture. The nine persons who preceded us on this platform have presented nine different views of computer science. For our organism, the machine, can be studied at many levels and from many sides. We are deeply honored to appear here today and to present yet another view, the one that has permeated the scientific work for which we have been

to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

The authors' research over the years has been supported in part by the Advanced Research Projects Agency of the Department of Defense (monitored by the Air Force Office of Scientific Research) and in part by the National Institutes of Mental Health.

Authors' address: Carnegie-Mellon University, Pittsburgh.

cited. We wish to speak of computer science as empirical inquiry.

Our view is only one of many; the previous lectures make that clear. However, even taken together the lectures fail to cover the whole scope of our science. Many fundamental aspects of it have not been represented in these ten awards. And if the time ever arrives, surely not soon, when the compass has been boxed, when computer science has been discussed from every side, it will be time to start the cycle again. For the hare as lecturer will have to make an annual sprint to overtake the cumulation of small, incremental gains that the tortoise of scientific and technical development has achieved in his steady march. Each year will create a new gap and call for a new sprint, for in science there is no final word.

Computer science is an empirical discipline. We would have called it an experimental science, but like astronomy, economics, and geology, some of its unique forms of observation and experience do not fit a narrow stereotype of the experimental method. None the less, they are experiments. Each new machine that is built is an experiment. Actually constructing the machine poses a question to nature; and we listen for the answer by observing the machine in operation and analyzing it by all analytical and measurement means available. Each new program that is built is an experiment. It poses a question to nature, and its behavior offers clues to an answer. Neither machines nor programs are black boxes; they are artifacts that have been designed, both hardware and software, and we can open them up and look inside. We can relate their structure to their behavior and draw many lessons from a single experiment. We don't have to build 100 copies of, say, a theorem prover, to demonstrate statistically that it has not overcome the combinatorial explosion of search in the way hoped for. Inspection of the program in the light of a few runs reveals the flaw and lets us proceed to the next attempt.

We build computers and programs for many reasons. We build them to serve society and as tools for carrying out the economic tasks of society. But as basic scientists we build machines and programs as a way of discovering new phenomena and analyzing phenomena we already know about. Society often becomes confused about this, believing that computers and programs are to be constructed only for the economic use that can be made of them (or as intermediate items in a developmental sequence leading to such use). It needs to understand that the phenomena surrounding computers are deep and obscure, requiring much experimentation to assess their nature. It needs to understand that, as in any

science, the gains that accrue from such experimentation and understanding pay off in the permanent acquisition of new techniques; and that it is these techniques that will create the instruments to help society in achieving its goals.

Our purpose here, however, is not to plead for understanding from an outside world. It is to examine one aspect of our science, the development of new basic understanding by empirical inquiry. This is best done by illustrations. We will be pardoned if, presuming upon the occasion, we choose our examples from the area of our own research. As will become apparent, these examples involve the whole development of artificial intelligence, especially in its early years. They rest on much more than our own personal contributions. And even where we have made direct contributions, this has been done in cooperation with others. Our collaborators have included especially Cliff Shaw, with whom we formed a team of three through the exciting period of the late fifties. But we have also worked with a great many colleagues and students at Carnegie-Mellon University.

Time permits taking up just two examples. The first is the development of the notion of a symbolic system. The second is the development of the notion of heuristic search. Both conceptions have deep significance for understanding how information is processed and how intelligence is achieved. However, they do not come close to exhausting the full scope of artificial intelligence, though they seem to us to be useful for exhibiting the nature of fundamental knowledge in this part of computer science.

I. Symbols and Physical Symbol Systems

One of the fundamental contributions to knowledge of computer science has been to explain, at a rather basic level, what symbols are. This explanation is a scientific proposition about Nature. It is empirically derived, with a long and gradual development.

Symbols lie at the root of intelligent action, which is, of course, the primary topic of artificial intelligence. For that matter, it is a primary question for all of computer science. For all information is processed by computers in the service of ends, and we measure the intelligence of a system by its ability to achieve stated ends in the face of variations, difficulties and complexities posed by the task environment. This general investment of computer science in attaining intelligence is obscured when the tasks being accomplished are

limited in scope, for then the full variations in the environment can be accurately foreseen. It becomes more obvious as we extend computers to more global, complex and knowledge-intensive tasks—as we attempt to make them our agents, capable of handling on their own the full contingencies of the natural world.

Our understanding of the systems requirements for intelligent action emerges slowly. It is composite, for no single elementary thing accounts for intelligence in all its manifestations. There is no “intelligence principle,” just as there is no “vital principle” that conveys by its very nature the essence of life. But the lack of a simple *deus ex machina* does not imply that there are no structural requirements for intelligence. One such requirement is the ability to store and manipulate symbols. To put the scientific question, we may paraphrase the title of a famous paper by Warren McCulloch [1961]: What is a symbol, that intelligence may use it, and intelligence, that it may use a symbol?

Laws of Qualitative Structure

All sciences characterize the essential nature of the systems they study. These characterizations are invariably qualitative in nature, for they set the terms within which more detailed knowledge can be developed. Their essence can often be captured in very short, very general statements. One might judge these general laws, due to their limited specificity, as making relatively little contribution to the sum of a science, were it not for the historical evidence that shows them to be results of the greatest importance.

The Cell Doctrine in Biology. A good example of a law of qualitative structure is the cell doctrine in biology, which states that the basic building block of all living organisms is the cell. Cells come in a large variety of forms, though they all have a nucleus surrounded by protoplasm, the whole encased by a membrane. But this internal structure was not, historically, part of the specification of the cell doctrine; it was subsequent specificity developed by intensive investigation. The cell doctrine can be conveyed almost entirely by the statement we gave above, along with some vague notions about what size a cell can be. The impact of this law on biology, however, has been tremendous, and the lost motion in the field prior to its gradual acceptance was considerable.

Plate Tectonics in Geology. Geology provides an interesting example of a qualitative structure law, interesting because it has gained acceptance in the last decade and so its rise in status is still fresh in memory. The

theory of plate tectonics asserts that the surface of the globe is a collection of huge plates—a few dozen in all—which move (at geological speeds) against, over, and under each other into the center of the earth, where they lose their identity. The movements of the plates account for the shapes and relative locations of the continents and oceans, for the areas of volcanic and earthquake activity, for the deep sea ridges, and so on. With a few additional particulars as to speed and size, the essential theory has been specified. It was of course not accepted until it succeeded in explaining a number of details, all of which hung together (e.g. accounting for flora, fauna, and stratification agreements between West Africa and Northeast South America). The plate tectonics theory is highly qualitative. Now that it is accepted, the whole earth seems to offer evidence for it everywhere, for we see the world in its terms.

The Germ Theory of Disease. It is little more than a century since Pasteur enunciated the germ theory of disease, a law of qualitative structure that produced a revolution in medicine. The theory proposes that most diseases are caused by the presence and multiplication in the body of tiny single-celled living organisms, and that contagion consists in the transmission of these organisms from one host to another. A large part of the elaboration of the theory consisted in identifying the organisms associated with specific diseases, describing them, and tracing their life histories. The fact that the law has many exceptions—that many diseases are not produced by germs—does not detract from its importance. The law tells us to look for a particular kind of cause; it does not insist that we will always find it.

The Doctrine of Atomism. The doctrine of atomism offers an interesting contrast to the three laws of qualitative structure we have just described. As it emerged from the work of Dalton and his demonstrations that the chemicals combined in fixed proportions, the law provided a typical example of qualitative structure: the elements are composed of small, uniform particles, differing from one element to another. But because the underlying species of atoms are so simple and limited in their variety, quantitative theories were soon formulated which assimilated all the general structure in the original qualitative hypothesis. With cells, tectonic plates, and germs, the variety of structure is so great that the underlying qualitative principle remains distinct, and its contribution to the total theory clearly discernible.

Conclusion. Laws of qualitative structure are seen everywhere in science. Some of our greatest scientific discoveries are to be found among them. As the examples illustrate, they often set the terms on which a whole science operates.

Physical Symbol Systems

Let us return to the topic of symbols, and define a *physical symbol system*. The adjective "physical" denotes two important features: (1) Such systems clearly obey the laws of physics—they are realizable by engineered systems made of engineered components; (2) although our use of the term "symbol" prefigures our intended interpretation, it is not restricted to human symbol systems.

A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). Thus, a symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way (such as one token being next to another). At any instant of time the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction. A physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbolic expressions themselves.

Two notions are central to this structure of expressions, symbols, and objects: designation and interpretation.

Designation. An expression designates an object if, given the expression, the system can either affect the object itself or behave in ways dependent on the object.

In either case, access to the object via the expression has been obtained, which is the essence of designation.

Interpretation. The system can interpret an expression if the expression designates a process and if, given the expression, the system can carry out the process.

Interpretation implies a special form of dependent action: given an expression the system can perform the indicated process, which is to say, it can evoke and execute its own processes from expressions that designate them.

A system capable of designation and interpretation, in the sense just indicated, must also meet a number of additional requirements, of completeness and closure. We will have space only to mention these briefly; all

of them are important and have far-reaching consequences.

(1) A symbol may be used to designate any expression whatsoever. That is, given a symbol, it is not prescribed a priori what expressions it can designate. This arbitrariness pertains only to symbols; the symbol tokens and their mutual relations determine what object is designated by a complex expression. (2) There exist expressions that designate every process of which the machine is capable. (3) There exist processes for creating any expression and for modifying any expression in arbitrary ways. (4) Expressions are stable; once created they will continue to exist until explicitly modified or deleted. (5) The number of expressions that the system can hold is essentially unbounded.

The type of system we have just defined is not unfamiliar to computer scientists. It bears a strong family resemblance to all general purpose computers. If a symbol manipulation language, such as LISP, is taken as defining a machine, then the kinship becomes truly brotherly. Our intent in laying out such a system is not to propose something new. Just the opposite: it is to show what is now known and hypothesized about systems that satisfy such a characterization.

We can now state a general scientific hypothesis—a law of qualitative structure for symbol systems:

The Physical Symbol System Hypothesis. A physical symbol system has the necessary and sufficient means for general intelligent action.

By "necessary" we mean that any system that exhibits general intelligence will prove upon analysis to be a physical symbol system. By "sufficient" we mean that any physical symbol system of sufficient size can be organized further to exhibit general intelligence. By "general intelligent action" we wish to indicate the same scope of intelligence as we see in human action: that in any real situation behavior appropriate to the ends of the system and adaptive to the demands of the environment can occur, within some limits of speed and complexity.

The Physical Symbol System Hypothesis clearly is a law of qualitative structure. It specifies a general class of systems within which one will find those capable of intelligent action.

This is an empirical hypothesis. We have defined a class of systems; we wish to ask whether that class accounts for a set of phenomena we find in the real world. Intelligent action is everywhere around us in the biological world, mostly in human behavior. It is a form of behavior we can recognize by its effects whether it is performed by humans or not. The hypothesis could indeed be false. Intelligent behavior is not so easy to produce that any system will exhibit it willy-nilly. Indeed, there are people whose analyses lead them to conclude either on philosophical or on scientific grounds that the hypothesis is false. Scientifically, one

can attack or defend it only by bringing forth empirical evidence about the natural world.

We now need to trace the development of this hypothesis and look at the evidence for it.

Development of the Symbol System Hypothesis

A physical symbol system is an instance of a universal machine. Thus the symbol system hypothesis implies that intelligence will be realized by a universal computer. However, the hypothesis goes far beyond the argument, often made on general grounds of physical determinism, that any computation that is realizable can be realized by a universal machine, provided that it is specified. For it asserts specifically that the intelligent machine is a symbol system, thus making a specific architectural assertion about the nature of intelligent systems. It is important to understand how this additional specificity arose.

Formal Logic. The roots of the hypothesis go back to the program of Frege and of Whitehead and Russell for formalizing logic: capturing the basic conceptual notions of mathematics in logic and putting the notions of proof and deduction on a secure footing. This effort culminated in mathematical logic—our familiar propositional, first-order, and higher-order logics. It developed a characteristic view, often referred to as the “symbol game.” Logic, and by incorporation all of mathematics, was a game played with meaningless tokens according to certain purely syntactic rules. All meaning had been purged. One had a mechanical, though permissive (we would now say nondeterministic), system about which various things could be proved. Thus progress was first made by walking away from all that seemed relevant to meaning and human symbols. We could call this the stage of formal symbol manipulation.

This general attitude is well reflected in the development of information theory. It was pointed out time and again that Shannon had defined a system that was useful only for communication and selection, and which had nothing to do with meaning. Regrets were expressed that such a general name as “information theory” had been given to the field, and attempts were made to rechristen it as “the theory of selective information”—to no avail, of course.

Turing Machines and the Digital Computer. The development of the first digital computers and of automata theory, starting with Turing’s own work in the ’30s, can be treated together. They agree in their view of what is essential. Let us use Turing’s own model, for it shows the features well.

A Turing machine consists of two memories: an unbounded tape and a finite state control. The tape holds data, i.e. the famous zeroes and ones. The machine has a very small set of proper operations—read, write, and scan operations—on the tape. The read operation is not a data operation, but provides conditional

branching to a control state as a function of the data under the read head. As we all know, this model contains the essentials of all computers, in terms of what they can do, though other computers with different memories and operations might carry out the same computations with different requirements of space and time. In particular, the model of a Turing machine contains within it the notions both of what cannot be computed and of universal machines—computers that can do anything that can be done by any machine.

We should marvel that two of our deepest insights into information processing were achieved in the thirties, before modern computers came into being. It is a tribute to the genius of Alan Turing. It is also a tribute to the development of mathematical logic at the time, and testimony to the depth of computer science’s obligation to it. Concurrently with Turing’s work appeared the work of the logicians Emil Post and (independently) Alonzo Church. Starting from independent notions of logistic systems (Post productions and recursive functions, respectively) they arrived at analogous results on undecidability and universality—results that were soon shown to imply that all three systems were equivalent. Indeed, the convergence of all these attempts to define the most general class of information processing systems provides some of the force of our conviction that we have captured the essentials of information processing in these models.

In none of these systems is there, on the surface, a concept of the symbol as something that *designates*. The data are regarded as just strings of zeroes and ones—indeed that data be inert is essential to the reduction of computation to physical process. The finite state control system was always viewed as a small controller, and logical games were played to see how small a state system could be used without destroying the universality of the machine. No games, as far as we can tell, were ever played to add new states dynamically to the finite control—to think of the control memory as holding the bulk of the system’s knowledge. What was accomplished at this stage was half the principle of interpretation—showing that a machine could be run from a description. Thus, this is the stage of automatic formal symbol manipulation.

The Stored Program Concept. With the development of the second generation of electronic machines in the mid-forties (after the Eniac) came the stored program concept. This was rightfully hailed as a milestone, both conceptually and practically. Programs now can be data, and can be operated on as data. This capability is, of course, already implicit in the model of Turing: the descriptions are on the very same tape as the data. Yet the idea was realized only when machines acquired enough memory to make it practicable to locate actual programs in some internal place. After all, the Eniac had only twenty registers.

The stored program concept embodies the second

half of the interpretation principle, the part that says that the system's own data can be interpreted. But it does not yet contain the notion of designation--of the physical relation that underlies meaning.

List Processing. The next step, taken in 1956, was list processing. The contents of the data structures were now symbols, in the sense of our physical symbol system: patterns that designated, that had referents. Lists held addresses which permitted access to other lists--thus the notion of list structures. That this was a new view was demonstrated to us many times in the early days of list processing when colleagues would ask where the data were--that is, which list finally held the collections of bits that were the content of the system. They found it strange that there were no such bits, there were only symbols that designated yet other symbol structures.

List processing is simultaneously three things in the development of computer science. (1) It is the creation of a genuine dynamic memory structure in a machine that had heretofore been perceived as having fixed structure. It added to our ensemble of operations those that built and modified structure in addition to those that replaced and changed content. (2) It was an early demonstration of the basic abstraction that a computer consists of a set of data types and a set of operations proper to these data types, so that a computational system should employ whatever data types are appropriate to the application, independent of the underlying machine. (3) List processing produced a model of designation, thus defining symbol manipulation in the sense in which we use this concept in computer science today.

As often occurs, the practice of the time already anticipated all the elements of list processing: addresses are obviously used to gain access, the drum machines used linked programs (so called one-plus-one addressing), and so on. But the conception of list processing as an abstraction created a new world in which designation and dynamic symbolic structure were the defining characteristics. The embedding of the early list processing systems in languages (the IPLs, LISP) is often decried as having been a barrier to the diffusion of list processing techniques throughout programming practice; but it was the vehicle that held the abstraction together.

LISP. One more step is worth noting: McCarthy's creation of LISP in 1959-60 [McCarthy, 1960]. It completed the act of abstraction, lifting list structures out of their embedding in concrete machines, creating a new formal system with S-expressions, which could be shown to be equivalent to the other universal schemes of computation.

Conclusion. That the concept of the designating symbol and symbol manipulation does not emerge until the mid-fifties does not mean that the earlier steps were either inessential or less important. The total

concept is the join of computability, physical realizability (and by multiple technologies), universality, the symbolic representation of processes (i.e. interpretability), and, finally, symbolic structure and designation. Each of the steps provided an essential part of the whole.

The first step in this chain, authored by Turing, is theoretically motivated, but the others all have deep empirical roots. We have been led by the evolution of the computer itself. The stored program principle arose out of the experience with ENIAC. List processing arose out of the attempt to construct intelligent programs. It took its cue from the emergence of random access memories, which provided a clear physical realization of a designating symbol in the address. LISP arose out of the evolving experience with list processing.

The Evidence

We come now to the evidence for the hypothesis that physical symbol systems are capable of intelligent action, and that general intelligent action calls for a physical symbol system. The hypothesis is an empirical generalization and not a theorem. We know of no way of demonstrating the connection between symbol systems and intelligence on purely logical grounds. Lacking such a demonstration, we must look at the facts. Our central aim, however, is not to review the evidence in detail, but to use the example before us to illustrate the proposition that computer science is a field of empirical inquiry. Hence, we will only indicate what kinds of evidence there is, and the general nature of the testing process.

The notion of physical symbol system had taken essentially its present form by the middle of the 1950's, and one can date from that time the growth of artificial intelligence as a coherent subfield of computer science. The twenty years of work since then has seen a continuous accumulation of empirical evidence of two main varieties. The first addresses itself to the *sufficiency* of physical symbol systems for producing intelligence, attempting to construct and test specific systems that have such a capability. The second kind of evidence addresses itself to the *necessity* of having a physical symbol system wherever intelligence is exhibited. It starts with Man, the intelligent system best known to us, and attempts to discover whether his cognitive activity can be explained as the working of a physical symbol system. There are other forms of evidence, which we will comment upon briefly later, but these two are the important ones. We will consider them in turn. The first is generally called artificial intelligence, the second, research in cognitive psychology.

Constructing Intelligent Systems. The basic paradigm for the initial testing of the germ theory of disease was: identify a disease; then look for the germ. An analogous paradigm has inspired much of the research in artificial intelligence: identify a task domain calling for intelligence; then construct a program for a digital computer

that can handle tasks in that domain. The easy and well-structured tasks were looked at first: puzzles and games, operations research problems of scheduling and allocating resources, simple induction tasks. Scores, if not hundreds, of programs of these kinds have by now been constructed, each capable of some measure of intelligent action in the appropriate domain.

Of course intelligence is not an all-or-none matter, and there has been steady progress toward higher levels of performance in specific domains, as well as toward widening the range of those domains. Early chess programs, for example, were deemed successful if they could play the game legally and with some indication of purpose; a little later, they reached the level of human beginners; within ten or fifteen years, they began to compete with serious amateurs. Progress has been slow (and the total programming effort invested small) but continuous, and the paradigm of construct-and-test proceeds in a regular cycle—the whole research activity mimicking at a macroscopic level the basic generate-and-test cycle of many of the AI programs.

There is a steadily widening area within which intelligent action is attainable. From the original tasks, research has extended to building systems that handle and understand natural language in a variety of ways, systems for interpreting visual scenes, systems for hand-eye coordination, systems that design, systems that write computer programs, systems for speech understanding—the list is, if not endless, at least very long. If there are limits beyond which the hypothesis will not carry us, they have not yet become apparent. Up to the present, the rate of progress has been governed mainly by the rather modest quantity of scientific resources that have been applied and the inevitable requirement of a substantial system-building effort for each new major undertaking.

Much more has been going on, of course, than simply a piling up of examples of intelligent systems adapted to specific task domains. It would be surprising and unappealing if it turned out that the AI programs performing these diverse tasks had nothing in common beyond their being instances of physical symbol systems. Hence, there has been great interest in searching for mechanisms possessed of generality, and for common components among programs performing a variety of tasks. This search carries the theory beyond the initial symbol system hypothesis to a more complete characterization of the particular kinds of symbol systems that are effective in artificial intelligence. In the second section of this paper, we will discuss one example of a hypothesis at this second level of specificity: the heuristic search hypothesis.

The search for generality spawned a series of programs designed to separate out general problem-solving mechanisms from the requirements of particular task domains. The General Problem Solver (GPS) was perhaps the first of these; while among its descendants are such contemporary systems as PLANNER and

CONNIVER. The search for common components has led to generalized schemes of representation for goals and plans, methods for constructing discrimination nets, procedures for the control of tree search, pattern-matching mechanisms, and language-parsing systems. Experiments are at present under way to find convenient devices for representing sequences of time and tense, movement, causality and the like. More and more, it becomes possible to assemble large intelligent systems in a modular way from such basic components.

We can gain some perspective on what is going on by turning, again, to the analogy of the germ theory. If the first burst of research stimulated by that theory consisted largely in finding the germ to go with each disease, subsequent effort turned to learning what a germ was—to building on the basic qualitative law a new level of structure. In artificial intelligence, an initial burst of activity aimed at building intelligent programs for a wide variety of almost randomly selected tasks is giving way to more sharply targeted research aimed at understanding the common mechanisms of such systems.

The Modeling of Human Symbolic Behavior. The symbol system hypothesis implies that the symbolic behavior of man arises because he has the characteristics of a physical symbol system. Hence, the results of efforts to model human behavior with symbol systems become an important part of the evidence for the hypothesis, and research in artificial intelligence goes on in close collaboration with research in information processing psychology, as it is usually called.

The search for explanations of man's intelligent behavior in terms of symbol systems has had a large measure of success over the past twenty years; to the point where information processing theory is the leading contemporary point of view in cognitive psychology. Especially in the areas of problem solving, concept attainment, and long-term memory, symbol manipulation models now dominate the scene.

Research in information processing psychology involves two main kinds of empirical activity. The first is the conduct of observations and experiments on human behavior in tasks requiring intelligence. The second, very similar to the parallel activity in artificial intelligence, is the programming of symbol systems to model the observed human behavior. The psychological observations and experiments lead to the formulation of hypotheses about the symbolic processes the subjects are using, and these are an important source of the ideas that go into the construction of the programs. Thus, many of the ideas for the basic mechanisms of GPS were derived from careful analysis of the protocols that human subjects produced while thinking aloud during the performance of a problem-solving task.

The empirical character of computer science is nowhere more evident than in this alliance with psy-

chology. Not only are psychological experiments required to test the veridicality of the simulation models as explanations of the human behavior, but out of the experiments come new ideas for the design and construction of physical symbol systems.

Other Evidence. The principal body of evidence for the symbol system hypothesis that we have not considered is negative evidence: the absence of specific competing hypotheses as to how intelligent activity might be accomplished—whether by man or machine. Most attempts to build such hypotheses have taken place within the field of psychology. Here we have had a continuum of theories from the points of view usually labeled “behaviorism” to those usually labeled “Gestalt theory.” Neither of these points of view stands as a real competitor to the symbol system hypothesis, and this for two reasons. First, neither behaviorism nor Gestalt theory has demonstrated, or even shown how to demonstrate, that the explanatory mechanisms it postulates are sufficient to account for intelligent behavior in complex tasks. Second, neither theory has been formulated with anything like the specificity of artificial programs. As a matter of fact, the alternative theories are sufficiently vague so that it is not terribly difficult to give them information processing interpretations, and thereby assimilate them to the symbol system hypothesis.

Conclusion

We have tried to use the example of the Physical Symbol System Hypothesis to illustrate concretely that computer science is a scientific enterprise in the usual meaning of that term: that it develops scientific hypotheses which it then seeks to verify by empirical inquiry. We had a second reason, however, for choosing this particular example to illustrate our point. The Physical Symbol System Hypothesis is itself a substantial scientific hypothesis of the kind that we earlier dubbed “laws of qualitative structure.” It represents an important discovery of computer science, which if borne out by the empirical evidence, as in fact appears to be occurring, will have major continuing impact on the field.

We turn now to a second example, the role of search in intelligence. This topic, and the particular hypothesis about it that we shall examine, have also played a central role in computer science, in general, and artificial intelligence, in particular.

II. Heuristic Search

Knowing that physical symbol systems provide the matrix for intelligent action does not tell us how they accomplish this. Our second example of a law of qualitative structure in computer science addresses this latter question, asserting that symbol systems solve problems by using the processes of heuristic search.

This generalization, like the previous one, rests on empirical evidence, and has not been derived formally from other premises. However, we shall see in a moment that it does have some logical connection with the symbol system hypothesis, and perhaps we can look forward to formalization of the connection at some time in the future. Until that time arrives, our story must again be one of empirical inquiry. We will describe what is known about heuristic search and review the empirical findings that show how it enables action to be intelligent. We begin by stating this law of qualitative structure, the Heuristic Search Hypothesis.

Heuristic Search Hypothesis. The solutions to problems are represented as symbol structures. A physical symbol system exercises its intelligence in problem solving by search—that is, by generating and progressively modifying symbol structures until it produces a solution structure.

Physical symbol systems must use heuristic search to solve problems because such systems have limited processing resources; in a finite number of steps, and over a finite interval of time, they can execute only a finite number of processes. Of course that is not a very strong limitation, for all universal Turing machines suffer from it. We intend the limitation, however, in a stronger sense: we mean *practically* limited. We can conceive of systems that are not limited in a practical way, but are capable, for example, of searching in parallel the nodes of an exponentially expanding tree at a constant rate for each unit advance in depth. We will not be concerned here with such systems, but with systems whose computing resources are scarce relative to the complexity of the situations with which they are confronted. The restriction will not exclude any real symbol systems, in computer or man, in the context of real tasks. The fact of limited resources allows us, for most purposes, to view a symbol system as though it were a serial, one-process-at-a-time device. If it can accomplish only a small amount of processing in any short time interval, then we might as well regard it as doing things one at a time. Thus “limited resource symbol system” and “serial symbol system” are practically synonymous. The problem of allocating a scarce resource from moment to moment can usually be treated, if the moment is short enough, as a problem of scheduling a serial machine.

Problem Solving

Since ability to solve problems is generally taken as a prime indicator that a system has intelligence, it is natural that much of the history of artificial intelligence is taken up with attempts to build and understand problem-solving systems. Problem solving has been discussed by philosophers and psychologists for two millennia, in discourses dense with the sense of mystery. If you think there is nothing problematic or mysterious about a symbol system solving problems, then you are

a child of today, whose views have been formed since mid-century. Plato (and, by his account, Socrates) found difficulty understanding even how problems could be *entertained*, much less how they could be solved. Let me remind you of how he posed the conundrum in the *Meno*:

Meno: And how will you inquire, Socrates, into that which you know not? What will you put forth as the subject of inquiry? And if you find what you want, how will you ever know that this is what you did not know?

To deal with this puzzle, Plato invented his famous theory of recollection: when you think you are discovering or learning something, you are really just recalling what you already knew in a previous existence. If you find this explanation preposterous, there is a much simpler one available today, based upon our understanding of symbol systems. An approximate statement of it is:

To state a problem is to designate (1) a *test* for a class of symbol structures (solutions of the problem), and (2) a *generator* of symbol structures (potential solutions). To solve a problem is to generate a structure, using (2), that satisfies the test of (1).

We have a problem if we know what we want to do (the test), and if we don't know immediately how to do it (our generator does not immediately produce a symbol structure satisfying the test). A symbol system can state and solve problems (sometimes) because it can generate and test.

If that is all there is to problem solving, why not simply generate at once an expression that satisfies the test? This is, in fact, what we do when we wish and dream. "If wishes were horses, beggars might ride." But outside the world of dreams, it isn't possible. To know how we would test something, once constructed, does not mean that we know how to construct it—that we have any generator for doing so.

For example, it is well known what it means to "solve" the problem of playing winning chess. A simple test exists for noticing winning positions, the test for checkmate of the enemy King. In the world of dreams one simply generates a strategy that leads to checkmate for all counter strategies of the opponent. Alas, no generator that will do this is known to existing symbol systems (man or machine). Instead, good moves in chess are sought by generating various alternatives, and painstakingly evaluating them with the use of approximate, and often erroneous, measures that are supposed to indicate the likelihood that a particular line of play is on the route to a winning position. Move generators there are; winning move generators there are not.

Before there can be a move generator for a problem, there must be a problem space: a space of symbol

structures in which problem situations, including the initial and goal situations, can be represented. Move generators are processes for modifying one situation in the problem space into another. The basic characteristics of physical symbol systems guarantee that they can represent problem spaces and that they possess move generators. How, in any concrete situation they synthesize a problem space and move generators appropriate to that situation is a question that is still very much on the frontier of artificial intelligence research.

The task that a symbol system is faced with, then, when it is presented with a problem and a problem space, is to use its limited processing resources to generate possible solutions, one after another, until it finds one that satisfies the problem-defining test. If the system had some control over the order in which potential solutions were generated, then it would be desirable to arrange this order of generation so that actual solutions would have a high likelihood of appearing early. A symbol system would exhibit intelligence to the extent that it succeeded in doing this. Intelligence for a system with limited processing resources consists in making wise choices of what to do next.

Search in Problem Solving

During the first decade or so of artificial intelligence research, the study of problem solving was almost synonymous with the study of search processes. From our characterization of problems and problem solving, it is easy to see why this was so. In fact, it might be asked whether it could be otherwise. But before we try to answer that question, we must explore further the nature of search processes as it revealed itself during that decade of activity.

Extracting Information from the Problem Space. Consider a set of symbol structures, some small subset of which are solutions to a given problem. Suppose, further, that the solutions are distributed randomly through the entire set. By this we mean that no information exists that would enable any search generator to perform better than a random search. Then no symbol system could exhibit more intelligence (or less intelligence) than any other in solving the problem, although one might experience better luck than another.

A condition, then, for the appearance of intelligence is that the distribution of solutions be not entirely random, that the space of symbol structures exhibit at least some degree of order and pattern. A second condition is that pattern in the space of symbol structures be more or less detectable. A third condition is that the generator of potential solutions be able to behave differentially, depending on what pattern it detected. There must be information in the problem space, and the symbol system must be capable of extracting and using it. Let us look first at a very simple example, where the intelligence is easy to come by.

Consider the problem of solving a simple algebraic equation:

$$AX + B = CX + D$$

The test defines a solution as any expression of the form, $X = E$, such that $AE + B = CE + D$. Now one could use as generator any process that would produce numbers which could then be tested by substituting in the latter equation. We would not call this an intelligent generator.

Alternatively, one could use generators that would make use of the fact that the original equation can be modified—by adding or subtracting equal quantities from both sides, or multiplying or dividing both sides by the same quantity—without changing its solutions. But, of course, we can obtain even more information to guide the generator by comparing the original expression with the form of the solution, and making precisely those changes in the equation that leave its solution unchanged, while at the same time, bringing it into the desired form. Such a generator could notice that there was an unwanted CX on the right-hand side of the original equation, subtract it from both sides and collect terms again. It could then notice that there was an unwanted B on the left-hand side and subtract that. Finally, it could get rid of the unwanted coefficient ($A - C$) on the left-hand side by dividing.

Thus by this procedure, which now exhibits considerable intelligence, the generator produces successive symbol structures, each obtained by modifying the previous one; and the modifications are aimed at reducing the differences between the form of the input structure and the form of the test expression, while maintaining the other conditions for a solution.

This simple example already illustrates many of the main mechanisms that are used by symbol systems for intelligent problem solving. First, each successive expression is not generated independently, but is produced by modifying one produced previously. Second, the modifications are not haphazard, but depend upon two kinds of information. They depend on information that is constant over this whole class of algebra problems, and that is built into the structure of the generator itself: all modifications of expressions must leave the equation's solution unchanged. They also depend on information that changes at each step: detection of the differences in form that remain between the current expression and the desired expression. In effect, the generator incorporates some of the tests the solution must satisfy, so that expressions that don't meet these tests will never be generated. Using the first kind of information guarantees that only a tiny subset of all possible expressions is actually generated, but without losing the solution expression from this subset. Using the second kind of information arrives at the desired solution by a succession of approximations, employing a simple form of means-ends analysis to give direction to the search.

There is no mystery where the information that guided the search came from. We need not follow Plato in endowing the symbol system with a previous existence in which it already knew the solution. A moderately sophisticated generator-test system did the trick without invoking reincarnation.

Search Trees. The simple algebra problem may seem an unusual, even pathological, example of search. It is certainly not trial-and-error search, for though there were a few trials, there was no error. We are more accustomed to thinking of problem-solving search as generating lushly branching trees of partial solution possibilities which may grow to thousands, or even millions, of branches, before they yield a solution. Thus, if from each expression it produces, the generator creates B new branches, then the tree will grow as B^D , where D is its depth. The tree grown for the algebra problem had the peculiarity that its branchiness, B , equaled unity.

Programs that play chess typically grow broad search trees, amounting in some cases to a million branches or more. (Although this example will serve to illustrate our points about tree search, we should note that the purpose of search in chess is not to generate proposed solutions, but to evaluate (test) them.) One line of research into game-playing programs has been centrally concerned with improving the representation of the chess board, and the processes for making moves on it, so as to speed up search and make it possible to search larger trees. The rationale for this direction, of course, is that the deeper the dynamic search, the more accurate should be the evaluations at the end of it. On the other hand, there is good empirical evidence that the strongest human players, grandmasters, seldom explore trees of more than one hundred branches. This economy is achieved not so much by searching less deeply than do chess-playing programs, but by branching very sparsely and selectively at each node. This is only possible, without causing a deterioration of the evaluations, by having more of the selectivity built into the generator itself, so that it is able to select for generation just those branches that are very likely to yield important relevant information about the position.

The somewhat paradoxical-sounding conclusion to which this discussion leads is that search—successive generation of potential solution structures—is a fundamental aspect of a symbol system's exercise of intelligence in problem solving but that amount of search is not a measure of the amount of intelligence being exhibited. What makes a problem a problem is not that a large amount of search is required for its solution, but that a large amount *would* be required if a requisite level of intelligence were not applied. When the symbolic system that is endeavoring to solve a problem knows enough about what to do, it simply proceeds directly towards its goal; but whenever its knowledge becomes inadequate, when it enters terra incognita, it

is faced with the threat of going through large amounts of search before it finds its way again.

The potential for the exponential explosion of the search tree that is present in every scheme for generating problem solutions warns us against depending on the brute force of computers—even the biggest and fastest computers—as a compensation for the ignorance and unselectivity of their generators. The hope is still periodically ignited in some human breasts that a computer can be found that is fast enough, and that can be programmed cleverly enough, to play good chess by brute-force search. There is nothing known in theory about the game of chess that rules out this possibility. Empirical studies on the management of search in sizable trees with only modest results make this a much less promising direction than it was when chess was first chosen as an appropriate task for artificial intelligence. We must regard this as one of the important empirical findings of research with chess programs.

The Forms of Intelligence. The task of intelligence, then, is to avert the ever-present threat of the exponential explosion of search. How can this be accomplished? The first route, already illustrated by the algebra example, and by chess programs that only generate "plausible" moves for further analysis, is to build selectivity into the generator: to generate only structures that show promise of being solutions or of being along the path toward solutions. The usual consequence of doing this is to decrease the rate of branching, not to prevent it entirely. Ultimate exponential explosion is not avoided—save in exceptionally highly structured situations like the algebra example—but only postponed. Hence, an intelligent system generally needs to supplement the selectivity of its solution generator with other information-using techniques to guide search.

Twenty years of experience with managing tree search in a variety of task environments has produced a small kit of general techniques which is part of the equipment of every researcher in artificial intelligence today. Since these techniques have been described in general works like that of Nilsson [1971], they can be summarized very briefly here.

In serial heuristic search, the basic question always is: what shall be done next? In tree search, that question, in turn, has two components: (1) from what node in the tree shall we search next, and (2) what direction shall we take from that node? Information helpful in answering the first question may be interpreted as measuring the relative distance of different nodes from the goal. Best-first search calls for searching next from the node that appears closest to the goal. Information helpful in answering the second question—in what direction to search—is often obtained, as in the algebra example, by detecting specific differences between the current nodal structure and the goal structure described by the test of a solution, and selecting actions that are relevant to reducing these particular kinds of

differences. This is the technique known as means-ends analysis, which plays a central role in the structure of the General Problem Solver.

The importance of empirical studies as a source of general ideas in AI research can be demonstrated clearly by tracing the history, through large numbers of problem solving programs, of these two central ideas: best-first search and means-ends analysis. Rudiments of best-first search were already present, though unnamed, in the Logic Theorist in 1955. The General Problem Solver, embodying means-ends analysis, appeared about 1957—but combined it with modified depth-first search rather than best-first search. Chess programs were generally wedded, for reasons of economy of memory, to depth-first search, supplemented after about 1958 by the powerful alpha beta pruning procedure. Each of these techniques appears to have been reinvented a number of times, and it is hard to find general, task-independent theoretical discussions of problem solving in terms of these concepts until the middle or late 1960's. The amount of formal buttressing they have received from mathematical theory is still minuscule: some theorems about the reduction in search that can be secured from using the alpha-beta heuristic, a couple of theorems (reviewed by Nilsson [1971]) about shortest-path search, and some very recent theorems on best-first search with a probabilistic evaluation function.

"Weak" and "Strong" Methods. The techniques we have been discussing are dedicated to the control of exponential expansion rather than its prevention. For this reason, they have been properly called "weak methods"—methods to be used when the symbol system's knowledge or the amount of structure actually contained in the problem space are inadequate to permit search to be avoided entirely. It is instructive to contrast a highly structured situation, which can be formulated, say, as a linear programming problem, with the less structured situations of combinatorial problems like the traveling salesman problem or scheduling problems. ("Less structured" here refers to the insufficiency or nonexistence of relevant theory about the structure of the problem space.)

In solving linear programming problems, a substantial amount of computation may be required, but the search does not branch. Every step is a step along the way to a solution. In solving combinatorial problems or in proving theorems, tree search can seldom be avoided, and success depends on heuristic search methods of the sort we have been describing.

Not all streams of AI problem-solving research have followed the path we have been outlining. An example of a somewhat different point is provided by the work on theorem-proving systems. Here, ideas imported from mathematics and logic have had a strong influence on the direction of inquiry. For example, the use of heuristics was resisted when properties of com-

plteness could not be proved (a bit ironic, since most interesting mathematical systems are known to be undecidable). Since completeness can seldom be proved for best-first search heuristics, or for many kinds of selective generators, the effect of this requirement was rather inhibiting. When theorem-proving programs were continually incapacitated by the combinatorial explosion of their search trees, thought began to be given to selective heuristics, which in many cases proved to be analogues of heuristics used in general problem-solving programs. The set-of-support heuristic, for example, is a form of working backwards, adapted to the resolution theorem proving environment.

A Summary of the Experience. We have now described the workings of our second law of qualitative structure, which asserts that physical symbol systems solve problems by means of heuristic search. Beyond that, we have examined some subsidiary characteristics of heuristic search, in particular the threat that it always faces of exponential explosion of the search tree, and some of the means it uses to avert that threat. Opinions differ as to how effective heuristic search has been as a problem solving mechanism—the opinions depending on what task domains are considered and what criterion of adequacy is adopted. Success can be guaranteed by setting aspiration levels low—or failure by setting them high. The evidence might be summed up about as follows. Few programs are solving problems at “expert” professional levels. Samuel’s checker program and Feigenbaum and Lederberg’s DENDRAL are perhaps the best-known exceptions, but one could point also to a number of heuristic search programs for such operations research problem domains as scheduling and integer programming. In a number of domains, programs perform at the level of competent amateurs: chess, some theorem-proving domains, many kinds of games and puzzles. Human levels have not yet been nearly reached by programs that have a complex perceptual “front end”: visual scene recognizers, speech understanders, robots that have to maneuver in real space and time. Nevertheless, impressive progress has been made, and a large body of experience assembled about these difficult tasks.

We do not have deep theoretical explanations for the particular pattern of performance that has emerged. On empirical grounds, however, we might draw two conclusions. First, from what has been learned about human expert performance in tasks like chess, it is likely that any system capable of matching that performance will have to have access, in its memories, to very large stores of semantic information. Second, some part of the human superiority in tasks with a large perceptual component can be attributed to the special-purpose built-in parallel processing structure of the human eye and ear.

In any case, the quality of performance must neces-

sarily depend on the characteristics both of the problem domains and of the symbol systems used to tackle them. For most real-life domains in which we are interested, the domain structure has not proved sufficiently simple to yield (so far) theorems about complexity, or to tell us, other than empirically, how large real-world problems are in relation to the abilities of our symbol systems to solve them. That situation may change, but until it does, we must rely upon empirical explorations, using the best problem solvers we know how to build, as a principal source of knowledge about the magnitude and characteristics of problem difficulty. Even in highly structured areas like linear programming, theory has been much more useful in strengthening the heuristics that underlie the most powerful solution algorithms than in providing a deep analysis of complexity.

Intelligence Without Much Search

Our analysis of intelligence equated it with ability to extract and use information about the structure of the problem space, so as to enable a problem solution to be generated as quickly and directly as possible. New directions for improving the problem-solving capabilities of symbol systems can be equated, then, with new ways of extracting and using information. At least three such ways can be identified.

Nonlocal Use of Information. First, it has been noted by several investigators that information gathered in the course of tree search is usually only used *locally*, to help make decisions at the specific node where the information was generated. Information about a chess position, obtained by dynamic analysis of a subtree of continuations, is usually used to evaluate just that position, not to evaluate other positions that may contain many of the same features. Hence, the same facts have to be rediscovered repeatedly at different nodes of the search tree. Simply to take the information out of the context in which it arose and use it generally does not solve the problem, for the information may be valid only in a limited range of contexts. In recent years, a few exploratory efforts have been made to transport information from its context of origin to other appropriate contexts. While it is still too early to evaluate the power of this idea, or even exactly how it is to be achieved, it shows considerable promise. An important line of investigation that Berliner [1975] has been pursuing is to use causal analysis to determine the range over which a particular piece of information is valid. Thus if a weakness in a chess position can be traced back to the move that made it, then the same weakness can be expected in other positions descendant from the same move.

The HEARSAY speech understanding system has taken another approach to making information globally available. That system seeks to recognize speech strings by pursuing a parallel search at a number of different

levels: phonemic, lexical, syntactic, and semantic. As each of these searches provides and evaluates hypotheses, it supplies the information it has gained to a common "blackboard" that can be read by all the sources. This shared information can be used, for example, to eliminate hypotheses, or even whole classes of hypotheses, that would otherwise have to be searched by one of the processes. Thus, increasing our ability to use tree-search information nonlocally offers promise for raising the intelligence of problem-solving systems.

Semantic Recognition Systems. A second active possibility for raising intelligence is to supply the symbol system with a rich body of semantic information about the task domain it is dealing with. For example, empirical research on the skill of chess masters shows that a major source of the master's skill is stored information that enables him to recognize a large number of specific features and patterns of features on a chess board, and information that uses this recognition to propose actions appropriate to the features recognized. This general idea has, of course, been incorporated in chess programs almost from the beginning. What is new is the realization of the number of such patterns and associated information that may have to be stored for master-level play: something of the order of 50,000.

The possibility of substituting recognition for search arises because a particular, and especially a rare, pattern can contain an enormous amount of information, provided that it is closely linked to the structure of the problem space. When that structure is "irregular," and not subject to simple mathematical description, then knowledge of a large number of relevant patterns may be the key to intelligent behavior. Whether this is so in any particular task domain is a question more easily settled by empirical investigation than by theory. Our experience with symbol systems richly endowed with semantic information and pattern-recognizing capabilities for accessing it is still extremely limited.

The discussion above refers specifically to semantic information associated with a recognition system. Of course, there is also a whole large area of AI research on semantic information processing and the organization of semantic memories that falls outside the scope of the topics we are discussing in this paper.

Selecting Appropriate Representations. A third line of inquiry is concerned with the possibility that search can be reduced or avoided by selecting an appropriate problem space. A standard example that illustrates this possibility dramatically is the mutilated checkerboard problem. A standard 64 square checkerboard can be covered exactly with 32 tiles, each a 1×2 rectangle covering exactly two squares. Suppose, now, that we cut off squares at two diagonally opposite corners of the checkerboard, leaving a total of 62 squares. Can this mutilated board be covered exactly with 31 tiles? With (literally) heavenly patience, the impossibility of achieving such a covering can be demonstrated by

trying all possible arrangements. The alternative, for those with less patience, and more intelligence, is to observe that the two diagonally opposite corners of a checkerboard are of the same color. Hence, the mutilated checkerboard has two less squares of one color than of the other. But each tile covers one square of one color and one square of the other, and any set of tiles must cover the same number of squares of each color. Hence, there is no solution. How can a symbol system discover this simple inductive argument as an alternative to a hopeless attempt to solve the problem by search among all possible coverings? We would award a system that found the solution high marks for intelligence.

Perhaps, however, in posing this problem we are not escaping from search processes. We have simply displaced the search from a space of possible problem solutions to a space of possible representations. In any event, the whole process of moving from one representation to another, and of discovering and evaluating representations, is largely unexplored territory in the domain of problem-solving research. The laws of qualitative structure governing representations remain to be discovered. The search for them is almost sure to receive considerable attention in the coming decade.

Conclusion

That is our account of symbol systems and intelligence. It has been a long road from Plato's *Meno* to the present, but it is perhaps encouraging that most of the progress along that road has been made since the turn of the twentieth century, and a large fraction of it since the midpoint of the century. Thought was still wholly intangible and ineffable until modern formal logic interpreted it as the manipulation of formal tokens. And it seemed still to inhabit mainly the heaven of Platonic ideals, or the equally obscure spaces of the human mind, until computers taught us how symbols could be processed by machines. A.M. Turing, whom we memorialize this morning, made his great contributions at the mid-century crossroads of these developments that led from modern logic to the computer.

Physical Symbol Systems. The study of logic and computers has revealed to us that intelligence resides in physical symbol systems. This is computer sciences's most basic law of qualitative structure.

Symbol systems are collections of patterns and processes, the latter being capable of producing, destroying and modifying the former. The most important properties of patterns is that they can designate objects, processes, or other patterns, and that, when they designate processes, they can be interpreted. Interpretation means carrying out the designated process. The two most significant classes of symbol systems with which we are acquainted are human beings and computers.

Our present understanding of symbol systems grew, as indicated earlier, through a sequence of stages. Formal logic familiarized us with symbols, treated syntactically, as the raw material of thought, and with the idea of manipulating them according to carefully defined formal processes. The Turing machine made the syntactic processing of symbols truly machine-like, and affirmed the potential universality of strictly defined symbol systems. The stored-program concept for computers reaffirmed the interpretability of symbols, already implicit in the Turing machine. List processing brought to the forefront the denotational capacities of symbols, and defined symbol processing in ways that allowed independence from the fixed structure of the underlying physical machine. By 1956 all of these concepts were available, together with hardware for implementing them. The study of the intelligence of symbol systems, the subject of artificial intelligence, could begin.

Heuristic Search. A second law of qualitative structure for AI is that symbol systems solve problems by generating potential solutions and testing them, that is, by searching. Solutions are usually sought by creating symbolic expressions and modifying them sequentially until they satisfy the conditions for a solution. Hence symbol systems solve problems by searching. Since they have finite resources, the search cannot be carried out all at once, but must be sequential. It leaves behind it either a single path from starting point to goal or, if correction and backup are necessary, a whole tree of such paths.

Symbol systems cannot appear intelligent when they are surrounded by pure chaos. They exercise intelligence by extracting information from a problem domain and using that information to guide their search, avoiding wrong turns and circuitous bypaths. The problem domain must contain information, that is, some degree of order and structure, for the method to work. The paradox of the *Meno* is solved by the observation that information may be remembered, but new information may also be extracted from the domain that the symbols designate. In both cases, the ultimate source of the information is the task domain.

The Empirical Base. Artificial intelligence research is concerned with how symbol systems must be organized in order to behave intelligently. Twenty years of work in the area has accumulated a considerable body of knowledge, enough to fill several books (it already has), and most of it in the form of rather concrete experience about the behavior of specific classes of symbol systems in specific task domains. Out of this experience, however, there have also emerged some generalizations, cutting across task domains and systems, about the general characteristics of intelligence and its methods of implementation.

We have tried to state some of these generalizations this morning. They are mostly qualitative rather than

mathematical. They have more the flavor of geology or evolutionary biology than the flavor of theoretical physics. They are sufficiently strong to enable us today to design and build moderately intelligent systems for a considerable range of task domains, as well as to gain a rather deep understanding of how human intelligence works in many situations.

What Next? In our account today, we have mentioned open questions as well as settled ones; there are many of both. We see no abatement of the excitement of exploration that has surrounded this field over the past quarter century. Two resource limits will determine the rate of progress over the next such period. One is the amount of computing power that will be available. The second, and probably the more important, is the number of talented young computer scientists who will be attracted to this area of research as the most challenging they can tackle.

A.M. Turing concluded his famous paper on "Computing Machinery and Intelligence" with the words:

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Many of the things Turing saw in 1950 that needed to be done have been done, but the agenda is as full as ever. Perhaps we read too much into his simple statement above, but we like to think that in it Turing recognized the fundamental truth that all computer scientists instinctively know. For all physical symbol systems, condemned as we are to serial search of the problem environment, the critical question is always: What to do next?

References

- Berliner, H. [1975]. Chess as problem solving: the development of a tactics analyzer. Ph.D. Th., Computer Sci. Dep., Carnegie-Mellon U. (unpublished).
- McCarthy, J. [1960]. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM* 3, 4 (April 1960), 184-195.
- McCulloch, W.S. [1961]. What is a number, that a man may know it, and a man, that he may know a number. *General Semantics Bulletin* Nos. 26 and 27 (1961), 7-18.
- Nilsson, N.J. [1971]. *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, New York.
- Turing, A.M. [1950]. Computing machinery and intelligence. *Mind* 59 (Oct. 1950), 433-460.

The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 17. In introducing the recipient, Jean E. Sammet, Chairman of the Awards Committee, made the following comments and read a portion of the final citation. The full announcement is in the September 1977 issue of *Communications*, page 681.

"Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B is for Backus, and the other letters are explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award."

The short form of his citation is for 'profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specifications of programming languages.'

The most significant part of the full citation is as follows:

'... Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical com-

putations called Fortran. This same group designed the first system to translate Fortran programs into machine language. They employed novel optimizing techniques to generate fast machine-language programs. Many other compilers for the language were developed, first on IBM machines, and later on virtually every make of computer. Fortran was adopted as a U.S. national standard in 1966.'

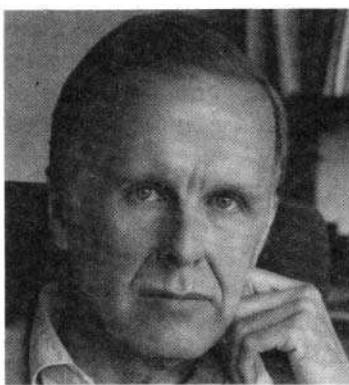
During the latter part of the 1950s, Backus served on the international committees which developed Algol 58 and a later version, Algol 60. The language Algol, and its derivative compilers, received broad acceptance in Europe as a means for developing programs and as a formal means of publishing the algorithms on which the programs are based.

In 1959, Backus presented a paper at the UNESCO conference in Paris on the syntax and semantics of a proposed international algebraic language. In this paper, he was the first to employ a formal technique for specifying the syntax of programming languages. The formal notation became known as BNF—standing for "Backus Normal Form," or "Backus Naur Form" to recognize the further contributions by Peter Naur of Denmark.

Thus, Backus has contributed strongly both to the pragmatic world of problem-solving on computers and to the theoretical world existing at the interface between artificial languages and computational linguistics. Fortran remains one of the most widely used programming languages in the world. Almost all programming languages are now described with some type of formal syntactic definition.'

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.
© 1978 ACM 0001-0782/78/0800-0613 \$00.75

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms. This algebra can be used to transform programs and to solve equations whose “unknowns” are programs in much the same way one transforms equations in high school algebra. These transformations are given by algebraic laws and are carried out in the same language in which programs are written. Combining forms are chosen not only for their programming power but also for the power of their associated algebraic laws. General theorems of the algebra give the detailed behavior and termination conditions for large classes of programs.

A new class of computing systems uses the functional programming style both in its programming language and in its state transition rules. Unlike von Neumann languages, these systems have semantics loosely coupled to states—only one state transition occurs per major computation.

Key Words and Phrases: functional programming, algebra of programs, combining forms, functional forms, programming languages, von Neumann computers, von Neumann languages, models of computing systems, applicative computing systems, applicative state transition systems, program transformation, program correctness, program termination, metacomposition

CR Categories: 4.20, 4.29, 5.20, 5.24, 5.26

Introduction

I deeply appreciate the honor of the ACM invitation to give the 1977 Turing Lecture and to publish this account of it with the details promised in the lecture. Readers wishing to see a summary of this paper should turn to Section 16, the last section.

1. Conventional Programming Languages: Fat and Flabby

Programming languages appear to be in trouble. Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more. Some languages have manuals exceeding 500 pages; others cram a complex description into shorter manuals by using dense formalisms. The Department of Defense has current plans for a committee-designed language standard that could require a manual as long as 1,000 pages. Each new language claims new and fashionable features, such as strong typing or structured control statements, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them.

Since large increases in size bring only small increases in power, smaller, more elegant languages such as Pascal continue to be popular. But there is a desperate need for a powerful methodology to help us think about pro-

grams, and no conventional language even begins to meet that need. In fact, conventional languages create unnecessary confusion in the way we think about programs.

For twenty years programming languages have been steadily progressing toward their present condition of obesity; as a result, the study and invention of programming languages has lost much of its excitement. Instead, it is now the province of those who prefer to work with thick compendia of details rather than wrestle with new ideas. Discussions about programming languages often resemble medieval debates about the number of angels that can dance on the head of a pin instead of exciting contests between fundamentally differing concepts.

Many creative computer scientists have retreated from inventing languages to inventing tools for describing them. Unfortunately, they have been largely content to apply their elegant new tools to studying the warts and moles of existing languages. After examining the appalling type structure of conventional languages, using the elegant tools developed by Dana Scott, it is surprising that so many of us remain passively content with that structure instead of energetically searching for new ones.

The purpose of this article is twofold; first, to suggest that basic defects in the framework of conventional languages make their expressive weakness and their cancerous growth inevitable, and second, to suggest some alternate avenues of exploration toward the design of new kinds of languages.

2. Models of Computing Systems

Underlying every programming language is a model of a computing system that its programs control. Some models are pure abstractions, some are represented by hardware, and others by compiling or interpretive programs. Before we examine conventional languages more closely, it is useful to make a brief survey of existing models as an introduction to the current universe of alternatives. Existing models may be crudely classified by the criteria outlined below.

2.1 Criteria for Models

2.1.1 Foundations. Is there an elegant and concise mathematical description of the model? Is it useful in proving helpful facts about the behavior of the model? Or is the model so complex that its description is bulky and of little mathematical use?

2.1.2 History sensitivity. Does the model include a notion of storage, so that one program can save information that can affect the behavior of a later program? That is, is the model history sensitive?

2.1.3 Type of semantics. Does a program successively transform states (which are not programs) until a terminal state is reached (state-transition semantics)? Are states simple or complex? Or can a “program” be successively reduced to simpler “programs” to yield a final

"normal form program," which is the result (reduction semantics)?

2.1.4 Clarity and conceptual usefulness of programs. Are programs of the model clear expressions of a process or computation? Do they embody concepts that help us to formulate and reason about processes?

2.2 Classification of Models

Using the above criteria we can crudely characterize three classes of models for computing systems—simple operational models, applicative models, and von Neumann models.

2.2.1 Simple operational models. Examples: Turing machines, various automata. *Foundations:* concise and useful. *History sensitivity:* have storage, are history sensitive. *Semantics:* state transition with very simple states. *Program clarity:* programs unclear and conceptually not helpful.

2.2.2 Applicative models. Examples: Church's lambda calculus [5], Curry's system of combinators [6], pure Lisp [17], functional programming systems described in this paper. *Foundations:* concise and useful. *History sensitivity:* no storage, not history sensitive. *Semantics:* reduction semantics, no states. *Program clarity:* programs can be clear and conceptually useful.

2.2.3 Von Neumann models. Examples: von Neumann computers, conventional programming languages. *Foundations:* complex, bulky, not useful. *History sensitivity:* have storage, are history sensitive. *Semantics:* state transition with complex states. *Program clarity:* programs can be moderately clear, are not very useful conceptually.

The above classification is admittedly crude and debatable. Some recent models may not fit easily into any of these categories. For example, the data-flow languages developed by Arvind and Gostelow [1], Dennis [7], Kosinski [13], and others partly fit the class of simple operational models, but their programs are clearer than those of earlier models in the class and it is perhaps possible to argue that some have reduction semantics. In any event, this classification will serve as a crude map of the territory to be discussed. We shall be concerned only with applicative and von Neumann models.

3. Von Neumann Computers

In order to understand the problems of conventional programming languages, we must first examine their intellectual parent, the von Neumann computer. What is a von Neumann computer? When von Neumann and others conceived it over thirty years ago, it was an elegant, practical, and unifying idea that simplified a number of engineering and programming problems that existed then. Although the conditions that produced its architecture have changed radically, we nevertheless still identify the notion of "computer" with this thirty year old concept.

In its simplest form a von Neumann computer has

three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the *von Neumann bottleneck*. The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear.

Ironically, a large part of the traffic in the bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Before a word can be sent through the tube its address must be in the CPU; hence it must either be sent through the tube from the store or be generated by some CPU operation. If the address is sent from the store, then its address must either have been sent from the store or generated in the CPU, and so on. If, on the other hand, the address is generated in the CPU, it must be generated either by a fixed rule (e.g., "add 1 to the program counter") or by an instruction that was sent through the tube, in which case its address must have been sent ... and so on.

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it.

4. Von Neumann Languages

Conventional programming languages are basically high level, complex versions of the von Neumann computer. Our thirty year old belief that there is only one kind of computer is the basis of our belief that there is only one kind of programming language, the conventional—von Neumann—language. The differences between Fortran and Algol 68, although considerable, are less significant than the fact that both are based on the programming style of the von Neumann computer. Although I refer to conventional languages as "von Neumann languages" to take note of their origin and style, I do not, of course, blame the great mathematician for their complexity. In fact, some might say that I bear some responsibility for that problem.

Von Neumann programming languages use variables to imitate the computer's storage cells; control statements elaborate its jump and test instructions; and assignment statements imitate its fetching, storing, and arithmetic.

The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.

Consider a typical program; at its center are a number of assignment statements containing some subscripted variables. Each assignment statement produces a one-word result. The program must cause these statements to be executed many times, while altering subscript values, in order to make the desired overall change in the store, since it must be done one word at a time. The programmer is thus concerned with the flow of words through the assignment bottleneck as he designs the nest of control statements to cause the necessary repetitions.

Moreover, the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.

The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements of the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.

This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control.

Our fixation on von Neumann languages has continued the primacy of the von Neumann computer, and our dependency on it has made non-von Neumann languages uneconomical and has limited their development. The absence of full scale, effective programming styles founded on non-von Neumann principles has deprived designers of an intellectual foundation for new computer architectures. (For a brief discussion of that topic, see Section 15.)

Applicative computing systems' lack of storage and history sensitivity is the basic reason they have not provided a foundation for computer design. Moreover, most applicative systems employ the substitution operation of the lambda calculus as their basic operation. This operation is one of virtually unlimited power, but its complete and efficient realization presents great difficulties to the machine designer. Furthermore, in an effort to introduce storage and to improve their efficiency on von Neumann computers, applicative systems have tended to become engulfed in a large von Neumann system. For example, pure Lisp is often buried in large extensions with many von Neumann features. The resulting complex systems offer little guidance to the machine designer.

5. Comparison of von Neumann and Functional Programs

To get a more detailed picture of some of the defects of von Neumann languages, let us compare a conventional program for inner product with a functional one written in a simple language to be detailed further on.

5.1 A von Neumann Program for Inner Product

```
c := 0  
for i := 1 step 1 until n do  
    c := c + a[i]×b[i]
```

Several properties of this program are worth noting:

- a) Its statements operate on an invisible "state" according to complex rules.
- b) It is not hierarchical. Except for the right side of the assignment statement, it does not construct complex entities from simpler ones. (Larger programs, however, often do.)
- c) It is dynamic and repetitive. One must mentally execute it to understand it.
- d) It computes word-at-a-time by repetition (of the assignment) and by modification (of variable i).
- e) Part of the data, n, is in the program; thus it lacks generality and works only for vectors of length n.
- f) It names its arguments; it can only be used for vectors a and b. To become general, it requires a procedure declaration. These involve complex issues (e.g., call-by-name versus call-by-value).
- g) Its "housekeeping" operations are represented by symbols in scattered places (in the for statement and the subscripts in the assignment). This makes it impossible to consolidate housekeeping operations, the most common of all, into single, powerful, widely useful operators. Thus in programming those operations one must always start again at square one, writing "for i := ..." and "for j := ..." followed by assignment statements sprinkled with i's and j's.

5.2 A Functional Program for Inner Product

Def Innerproduct

$$= (\text{Insert } +) \circ (\text{ApplyToAll } \times) \circ \text{Transpose}$$

Or, in abbreviated form:

Def IP = (+)∘(α×)∘Trans.

Composition (\circ), Insert (/), and ApplyToAll (α) are *functional forms* that combine existing functions to form new ones. Thus $f \circ g$ is the function obtained by applying first g and then f , and αf is the function obtained by applying f to every *member* of the argument. If we write $f:x$ for the result of applying f to the object x , then we can explain each step in evaluating Innerproduct applied to the pair of vectors $\langle\langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle\rangle$ as follows:

$$\begin{aligned} \text{IP: } &\langle\langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle\rangle = \\ \text{Definition of IP} &\Rightarrow (+) \circ (\alpha \times) \circ \text{Trans: } \langle\langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle\rangle \\ \text{Effect of composition, } \circ &\Rightarrow (+) \circ ((\alpha \times) \circ \text{Trans: } \\ &\quad \langle\langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle\rangle)) \end{aligned}$$

Applying Transpose	$\Rightarrow (+)((\alpha x): <<1,6>, <2,5>, <3,4>>)$
Effect of ApplyToAll, α	$\Rightarrow (+): <x: <1,6>, x: <2,5>, x: <3,4>>$
Applying \times	$\Rightarrow (+): <6,10,12>$
Effect of Insert, /	$\Rightarrow +: <6, +: <10,12>>$
Applying +	$\Rightarrow +: <6,22>$
Applying + again	$\Rightarrow 28$

Let us compare the properties of this program with those of the von Neumann program.

a) It operates only on its arguments. There are no hidden states or complex transition rules. There are only two kinds of rules, one for applying a function to its argument, the other for obtaining the function denoted by a functional form such as composition, $f \circ g$, or ApplyToAll, αf , when one knows the functions f and g , the *parameters* of the forms.

b) It is hierarchical, being built from three simpler functions ($+$, \times , Trans) and three functional forms $f \circ g$, αf , and $/f$.

c) It is static and nonrepetitive, in the sense that its structure is helpful in understanding it without mentally executing it. For example, if one understands the action of the forms $f \circ g$ and αf , and of the functions \times and Trans, then one understands the action of $\alpha \times$ and of $(\alpha \times) \circ \text{Trans}$, and so on.

d) It operates on whole conceptual units, not words; it has three steps; no step is repeated.

e) It incorporates no data; it is completely general; it works for any pair of conformable vectors.

f) It does not name its arguments; it can be applied to any pair of vectors without any procedure declaration or complex substitution rules.

g) It employs housekeeping forms and functions that are generally useful in many other programs; in fact, only $+$ and \times are not concerned with housekeeping. These forms and functions can combine with others to create higher level housekeeping operators.

Section 14 sketches a kind of system designed to make the above functional style of programming available in a history-sensitive system with a simple framework, but much work remains to be done before the above applicative style can become the basis for elegant and practical programming languages. For the present, the above comparison exhibits a number of serious flaws in von Neumann programming languages and can serve as a starting point in an effort to account for their present fat and flabby condition.

6. Language Frameworks versus Changeable Parts

Let us distinguish two parts of a programming language. First, its *framework* which gives the overall rules of the system, and second, its *changeable parts*, whose existence is anticipated by the framework but whose particular behavior is not specified by it. For example, the *for* statement, and almost all other statements, are part of Algol's framework but library functions and user-defined procedures are changeable parts. Thus the framework of a language describes its fixed features and

provides a general environment for its changeable features.

Now suppose a language had a small framework which could accommodate a great variety of powerful features entirely as changeable parts. Then such a framework could support many different features and styles without being changed itself. In contrast to this pleasant possibility, von Neumann languages always seem to have an immense framework and very limited changeable parts. What causes this to happen? The answer concerns two problems of von Neumann languages.

The first problem results from the von Neumann style of word-at-a-time programming, which requires that words flow back and forth to the state, just like the flow through the von Neumann bottleneck. Thus a von Neumann language must have a semantics closely coupled to the state, in which every detail of a computation changes the state. The consequence of this semantics closely coupled to states is that every detail of every feature must be built into the state and its transition rules.

Thus every feature of a von Neumann language must be spelled out in stupefying detail in its framework. Furthermore, many complex features are needed to prop up the basically weak word-at-a-time style. The result is the inevitable rigid and enormous framework of a von Neumann language.

7. Changeable Parts and Combining Forms

The second problem of von Neumann languages is that their changeable parts have so little expressive power. Their gargantuan size is eloquent proof of this; after all, if the designer knew that all those complicated features, which he now builds into the framework, could be added later on as changeable parts, he would not be so eager to build them into the framework.

Perhaps the most important element in providing powerful changeable parts in a language is the availability of combining forms that can be generally used to build new procedures from old ones. Von Neumann languages provide only primitive combining forms, and the von Neumann framework presents obstacles to their full use.

One obstacle to the use of combining forms is the split between the expression world and the statement world in von Neumann languages. Functional forms naturally belong to the world of expressions; but no matter how powerful they are they can only build expressions that produce a one-word result. And it is in the statement world that these one-word results must be combined into the overall result. Combining single words is not what we really should be thinking about, but it is a large part of programming any task in von Neumann languages. To help assemble the overall result from single words these languages provide some primitive combining forms in the statement world—the *for*, *while*, and *if-then-else* statements—but the split between the

two worlds prevents the combining forms in either world from attaining the full power they can achieve in an undivided world.

A second obstacle to the use of combining forms in von Neumann languages is their use of elaborate naming conventions, which are further complicated by the substitution rules required in calling procedures. Each of these requires a complex mechanism to be built into the framework so that variables, subscripted variables, pointers, file names, procedure names, call-by-value formal parameters, call-by-name formal parameters, and so on, can all be properly interpreted. All these names, conventions, and rules interfere with the use of simple combining forms.

8. APL versus Word-at-a-Time Programming

Since I have said so much about word-at-a-time programming, I must now say something about APL [12]. We owe a great debt to Kenneth Iverson for showing us that there are programs that are neither word-at-a-time nor dependent on lambda expressions, and for introducing us to the use of new functional forms. And since APL assignment statements can store arrays, the effect of its functional forms is extended beyond a single assignment.

Unfortunately, however, APL still splits programming into a world of expressions and a world of statements. Thus the effort to write one-line programs is partly motivated by the desire to stay in the more orderly world of expressions. APL has exactly three functional forms, called inner product, outer product, and reduction. These are sometimes difficult to use, there are not enough of them, and their use is confined to the world of expressions.

Finally, APL semantics is still too closely coupled to states. Consequently, despite the greater simplicity and power of the language, its framework has the complexity and rigidity characteristic of von Neumann languages.

9. Von Neumann Languages Lack Useful Mathematical Properties

So far we have discussed the gross size and inflexibility of von Neumann languages; another important defect is their lack of useful mathematical properties and the obstacles they present to reasoning about programs. Although a great amount of excellent work has been published on proving facts about programs, von Neumann languages have almost no properties that are helpful in this direction and have many properties that are obstacles (e.g., side effects, aliasing).

Denotational semantics [23] and its foundations [20, 21] provide an extremely helpful mathematical understanding of the domain and function spaces implicit in programs. When applied to an applicative language (such as that of the "recursive programs" of [16]), its

foundations provide powerful tools for describing the language and for proving properties of programs. When applied to a von Neumann language, on the other hand, it provides a precise semantic description and is helpful in identifying trouble spots in the language. But the complexity of the language is mirrored in the complexity of the description, which is a bewildering collection of productions, domains, functions, and equations that is only slightly more helpful in proving facts about programs than the reference manual of the language, since it is less ambiguous.

Axiomatic semantics [11] precisely restates the inelegant properties of von Neumann programs (i.e., transformations on states) as transformations on predicates. The word-at-a-time, repetitive game is not thereby changed, merely the playing field. The complexity of this axiomatic game of proving facts about von Neumann programs makes the successes of its practitioners all the more admirable. Their success rests on two factors in addition to their ingenuity: First, the game is restricted to small, weak subsets of full von Neumann languages that have states vastly simpler than real ones. Second, the new playing field (predicates and their transformations) is richer, more orderly and effective than the old (states and their transformations). But restricting the game and transferring it to a more effective domain does not enable it to handle real programs (with the necessary complexities of procedure calls and aliasing), nor does it eliminate the clumsy properties of the basic von Neumann style. As axiomatic semantics is extended to cover more of a typical von Neumann language, it begins to lose its effectiveness with the increasing complexity that is required.

Thus denotational and axiomatic semantics are descriptive formalisms whose foundations embody elegant and powerful concepts; but using them to describe a von Neumann language can not produce an elegant and powerful language any more than the use of elegant and modern machines to build an Edsel can produce an elegant and modern car.

In any case, proofs about programs use the language of logic, not the language of programming. Proofs talk about programs but cannot involve them directly since the axioms of von Neumann languages are so unusable. In contrast, many ordinary proofs are derived by algebraic methods. These methods require a language that has certain algebraic properties. Algebraic laws can then be used in a rather mechanical way to transform a problem into its solution. For example, to solve the equation

$$ax + bx = a + b$$

for x (given that $a+b \neq 0$), we mechanically apply the distributive, identity, and cancellation laws, in succession, to obtain

$$\begin{aligned} (a + b)x &= a + b \\ (a + b)x &= (a + b)1 \\ x &= 1. \end{aligned}$$

Thus we have proved that $x = 1$ without leaving the "language" of algebra. Von Neumann languages, with their grotesque syntax, offer few such possibilities for transforming programs.

As we shall see later, programs can be expressed in a language that has an associated algebra. This algebra can be used to transform programs and to solve some equations whose "unknowns" are programs, in much the same way one solves equations in high school algebra. Algebraic transformations and proofs use the language of the programs themselves, rather than the language of logic, which talks about programs.

10. What Are the Alternatives to von Neumann Languages?

Before discussing alternatives to von Neumann languages, let me remark that I regret the need for the above negative and not very precise discussion of these languages. But the complacent acceptance most of us give to these enormous, weak languages has puzzled and disturbed me for a long time. I am disturbed because that acceptance has consumed a vast effort toward making von Neumann languages fatter that might have been better spent in looking for new structures. For this reason I have tried to analyze some of the basic defects of conventional languages and show that those defects cannot be resolved unless we discover a new kind of language framework.

In seeking an alternative to conventional languages we must first recognize that a system cannot be history sensitive (permit execution of one program to affect the behavior of a subsequent one) unless the system has some kind of state (which the first program can change and the second can access). Thus a history-sensitive model of a computing system must have a state-transition semantics, at least in this weak sense. But this does *not* mean that every computation must depend heavily on a complex state, with many state changes required for each small part of the computation (as in von Neumann languages).

To illustrate some alternatives to von Neumann languages, I propose to sketch a class of history-sensitive computing systems, where each system: a) has a loosely coupled state-transition semantics in which a state transition occurs only once in a major computation; b) has a simply structured state and simple transition rules; c) depends heavily on an underlying applicative system both to provide the basic programming language of the system and to describe its state transitions.

These systems, which I call applicative state transition (or AST) systems, are described in Section 14. These simple systems avoid many of the complexities and weaknesses of von Neumann languages and provide for a powerful and extensive set of changeable parts. However, they are sketched only as crude examples of a vast area of non-von Neumann systems with various attractive properties. I have been studying this area for the

past three or four years and have not yet found a satisfying solution to the many conflicting requirements that a good language must resolve. But I believe this search has indicated a useful approach to designing non-von Neumann languages.

This approach involves four elements, which can be summarized as follows.

a) *A functional style of programming without variables.* A simple, informal functional programming (FP) system is described. It is based on the use of combining forms for building programs. Several programs are given to illustrate functional programming.

b) *An algebra of functional programs.* An algebra is described whose variables denote FP functional programs and whose "operations" are FP functional forms, the combining forms of FP programs. Some laws of the algebra are given. Theorems and examples are given that show how certain function expressions may be transformed into equivalent infinite expansions that explain the behavior of the function. The FP algebra is compared with algebras associated with the classical applicative systems of Church and Curry.

c) *A formal functional programming system.* A formal (FFP) system is described that extends the capabilities of the above informal FP systems. An FFP system is thus a precisely defined system that provides the ability to use the functional programming style of FP systems and their algebra of programs. FFP systems can be used as the basis for applicative state transition systems.

d) *Applicative state transition systems.* As discussed above. The rest of the paper describes these four elements, gives some brief remarks on computer design, and ends with a summary of the paper.

11. Functional Programming Systems (FP Systems)

11.1 Introduction

In this section we give an informal description of a class of simple applicative programming systems called functional programming (FP) systems, in which "programs" are simply functions without variables. The description is followed by some examples and by a discussion of various properties of FP systems.

An FP system is founded on the use of a fixed set of combining forms called functional forms. These, plus simple definitions, are the only means of building new functions from existing ones; they use no variables or substitution rules, and they become the operations of an associated algebra of programs. All the functions of an FP system are of one type: they map objects into objects and always take a single argument.

In contrast, a lambda-calculus based system is founded on the use of the lambda expression, with an associated set of substitution rules for variables, for building new functions. The lambda expression (with its substitution rules) is capable of defining all possible computable functions of all possible types and of any number of arguments. This freedom and power has its

disadvantages as well as its obvious advantages. It is analogous to the power of unrestricted control statements in conventional languages: with unrestricted freedom comes chaos. If one constantly invents new combining forms to suit the occasion, as one can in the lambda calculus, one will not become familiar with the style or useful properties of the few combining forms that are adequate for all purposes. Just as structured programming eschews many control statements to obtain programs with simpler structure, better properties, and uniform methods for understanding their behavior, so functional programming eschews the lambda expression, substitution, and multiple function types. It thereby achieves programs built with familiar functional forms with known useful properties. These programs are so structured that their behavior can often be understood and proven by mechanical use of algebraic techniques similar to those used in solving high school algebra problems.

Functional forms, unlike most programming constructs, need not be chosen on an ad hoc basis. Since they are the operations of an associated algebra, one chooses only those functional forms that not only provide powerful programming constructs, but that also have attractive algebraic properties: one chooses them to maximize the strength and utility of the algebraic laws that relate them to other functional forms of the system.

In the following description we shall be imprecise in not distinguishing between (a) a function symbol or expression and (b) the function it denotes. We shall indicate the symbols and expressions used to denote functions by example and usage. Section 13 describes a formal extension of FP systems (FFP systems); they can serve to clarify any ambiguities about FP systems.

11.2 Description

An FP system comprises the following:

- 1) a set O of *objects*;
- 2) a set F of *functions* f that map objects into objects;
- 3) an operation, *application*;
- 4) a set F of *functional forms*; these are used to combine existing functions, or objects, to form new functions in F ;
- 5) a set D of *definitions* that define some functions in F and assign a name to each.

What follows is an informal description of each of the above entities with examples.

11.2.1 Objects, O. An *object* x is either an *atom*, a *sequence* $\langle x_1, \dots, x_n \rangle$ whose *elements* x_i are objects, or \perp ("bottom" or "undefined"). Thus the choice of a set A of atoms determines the set of objects. We shall take A to be the set of nonnull strings of capital letters, digits, and special symbols not used by the notation of the FP system. Some of these strings belong to the class of atoms called "numbers." The atom ϕ is used to denote the empty sequence and is the only object which is both an atom and a sequence. The atoms T and F are used to denote "true" and "false."

There is one important constraint in the construction of objects: if x is a sequence with \perp as an element, then $x = \perp$. That is, the "sequence constructor" is " \perp -preserving." Thus no proper sequence has \perp as an element.

Examples of objects

$\perp \quad I.5 \quad \phi \quad AB3 \quad \langle AB, I, 2.3 \rangle$
 $\langle A, \langle\langle B \rangle, C \rangle, D \rangle \quad \langle A, \perp \rangle = \perp$

11.2.2 Application. An FP system has a single operation, application. If f is a function and x is an object, then $f:x$ is an *application* and denotes the object which is the result of applying f to x . f is the *operator* of the application and x is the *operand*.

Examples of applications

$+:\langle I, 2 \rangle = 3 \quad tl:\langle A, B, C \rangle = \langle B, C \rangle$
 $1:\langle A, B, C \rangle = A \quad 2:\langle A, B, C \rangle = B$

11.2.3 Functions, F. All functions f in F map objects into objects and are *bottom-preserving*: $f:\perp = \perp$, for all f in F . Every function in F is either *primitive*, that is, supplied with the system, or it is *defined* (see below), or it is a *functional form* (see below).

It is sometimes useful to distinguish between two cases in which $f:x=\perp$. If the computation for $f:x$ terminates and yields the object \perp , we say f is *undefined* at x , that is, f terminates but has no meaningful value at x . Otherwise we say f is *nonterminating* at x .

Examples of primitive functions

Our intention is to provide FP systems with widely useful and powerful primitive functions rather than weak ones that could then be used to define useful ones. The following examples define some typical primitive functions, many of which are used in later examples of programs. In the following definitions we use a variant of McCarthy's conditional expressions [17]; thus we write

$p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n; e_{n+1}$

instead of McCarthy's expression

$(p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n, T \rightarrow e_{n+1})$.

The following definitions are to hold for all objects x, x_i, y, y_i, z, z_i :

Selector functions

$1:x \equiv x = \langle x_1, \dots, x_n \rangle \rightarrow x_1; \perp$

and for any positive integer s

$s:x \equiv x = \langle x_1, \dots, x_n \rangle \& n \geq s \rightarrow x_s; \perp$

Thus, for example, $3:\langle A, B, C \rangle = C$ and $2:\langle A \rangle = \perp$. Note that the function symbols 1, 2, etc. are distinct from the atoms I , 2 , etc.

Tail

$tl:x \equiv x = \langle x_1 \rangle \rightarrow \phi;$
 $x = \langle x_1, \dots, x_n \rangle \& n \geq 2 \rightarrow \langle x_2, \dots, x_n \rangle; \perp$

Identity

$id:x \equiv x$

Atomatom: $x = x$ is an atom $\rightarrow T; x \neq \perp \rightarrow F; \perp$ **Equals**eq: $x = x = <y, z> \& y = z \rightarrow T; x = <y, z> \& y \neq z \rightarrow F; \perp$ **Null**null: $x = x = \phi \rightarrow T; x \neq \perp \rightarrow F; \perp$ **Reverse**reverse: $x = x = \phi \rightarrow \phi;$

$$x = <x_1, \dots, x_n> \rightarrow <x_n, \dots, x_1>; \perp$$

Distribute from left; distribute from rightdistl: $x = x = <y, \phi> \rightarrow \phi;$

$$x = <y, <z_1, \dots, z_n>> \rightarrow <<y, z_1>, \dots, <y, z_n>>; \perp$$

distr: $x = x = <\phi, y> \rightarrow \phi;$

$$x = <<y_1, \dots, y_n>, z> \rightarrow <<y_1, z>, \dots, <y_n, z>>; \perp$$

Lengthlength: $x = x = <x_1, \dots, x_n> \rightarrow n; x = \phi \rightarrow 0; \perp$ **Add, subtract, multiply, and divide**+: $x = x = <y, z> \& y, z \text{ are numbers} \rightarrow y + z; \perp$ -: $x = x = <y, z> \& y, z \text{ are numbers} \rightarrow y - z; \perp$ *: $x = x = <y, z> \& y, z \text{ are numbers} \rightarrow y \times z; \perp$ / $x = x = <y, z> \& y, z \text{ are numbers} \rightarrow y \div z; \perp$
(where $y \div 0 = \perp$)**Transpose**trans: $x = x = <\phi, \dots, \phi> \rightarrow \phi;$

$$x = <x_1, \dots, x_n> \rightarrow <y_1, \dots, y_m>; \perp$$

where

 $x_i = <x_{i1}, \dots, x_{im}>$ and

$$y_j = <x_{ij}, \dots, x_{in}>, 1 \leq i \leq n, 1 \leq j \leq m.$$

And, or, notand: $x = x = <T, T> \rightarrow T;$

$$x = <T, F> \vee x = <F, T> \vee x = <F, F> \rightarrow F; \perp$$

etc.

Append left; append rightapndl: $x = x = <y, \phi> \rightarrow <y>;$

$$x = <y, <z_1, \dots, z_n>> \rightarrow <y, z_1, \dots, z_n>; \perp$$

apndr: $x = x = <\phi, z> \rightarrow <z>;$

$$x = <<y_1, \dots, y_n>, z> \rightarrow <y_1, \dots, y_n, z>; \perp$$

Right selectors; Right taillr: $x = x = <x_1, \dots, x_n> \rightarrow x_n; \perp$ 2r: $x = x = <x_1, \dots, x_n> \& n \geq 2 \rightarrow x_{n-1}; \perp$

etc.

tlr: $x = x = <x_1> \rightarrow \phi;$

$$x = <x_1, \dots, x_n> \& n \geq 2 \rightarrow <x_1, \dots, x_{n-1}>; \perp$$

Rotate left; rotate rightrotl: $x = x = \phi \rightarrow \phi; x = <x_1> \rightarrow <x_1>;$

$$x = <x_1, \dots, x_n> \& n \geq 2 \rightarrow <x_2, \dots, x_n, x_1>; \perp$$

etc.

11.2.4 Functional forms, F. A functional form is an expression denoting a function; that function depends on the functions or objects which are the *parameters* of the expression. Thus, for example, if f and g are any functions, then $f \circ g$ is a functional form, the *composition* of f

and g , f and g are its parameters, and it denotes the function such that, for any object x ,

$$(f \circ g):x = f:(g:x).$$

Some functional forms may have objects as parameters. For example, for any object x , \bar{x} is a functional form, the *constant* function of x , so that for any object y

$$\bar{x}:y = y = \perp \rightarrow \perp; x.$$

In particular, \perp is the everywhere- \perp function.

Below we give some functional forms, many of which are used later in this paper. We use p, f , and g with and without subscripts to denote arbitrary functions; and x, x_1, \dots, x_n, y as arbitrary objects. Square brackets [...] are used to indicate the functional form for *construction*, which denotes a function, whereas pointed brackets <...> denote sequences, which are objects. Parentheses are used both in particular functional forms (e.g., in *condition*) and generally to indicate grouping.

Composition

$$(f \circ g):x = f:(g:x)$$

Construction

$[f_1, \dots, f_n]:x = <f_1:x, \dots, f_n:x>$ (Recall that since $<\dots, \perp, \dots> = \perp$ and all functions are \perp -preserving, so is $[f_1, \dots, f_n]$.)

Condition

$$(p \rightarrow f; g):x = (p:x) = T \rightarrow f:x; (p:x) = F \rightarrow g:x; \perp$$

Conditional *expressions* (used outside of FP systems to describe their functions) and the *functional form* condition are both identified by “ \rightarrow ”. They are quite different although closely related, as shown in the above definitions. But no confusion should arise, since the elements of a conditional expression all denote values, whereas the elements of the functional form condition all denote functions, never values. When no ambiguity arises we omit right-associated parentheses; we write, for example, $p_1 \rightarrow f_1; p_2 \rightarrow f_2; g$ for $(p_1 \rightarrow f_1; (p_2 \rightarrow f_2; g))$.

Constant (Here x is an object parameter.)

$$\bar{x}:y = y = \perp \rightarrow \perp; x$$

Insert

$$\begin{aligned} /f:x &= x = <x_1> \rightarrow x_1; x = <x_1, \dots, x_n> \& n \geq 2 \\ &\rightarrow f:<x_1, /f:<x_2, \dots, x_n>>; \perp \end{aligned}$$

If f has a unique right unit $u_f \neq \perp$, where $f:<x, u_f> \in \{x, \perp\}$ for all objects x , then the above definition is extended: $/f:\phi = u_f$. Thus

$$\begin{aligned} /+: <4, 5, 6> &= +: <4, +: <5, /+: <6>>> \\ &= +: <4, +: <5, 6>> = 15 \end{aligned}$$

$$/+: \phi = 0$$

Apply to all

$$\begin{aligned} af:x &= x = \phi \rightarrow \phi; \\ x = <x_1, \dots, x_n> &\rightarrow <f:x_1, \dots, f:x_n>; \perp \end{aligned}$$

Binary to unary (x is an object parameter)

$(\text{bu } f \ x) : y \equiv f : <x, y>$

Thus

$(\text{bu } I) : x = I + x$

While

$(\text{while } p \ f) : x \equiv p : x = T \rightarrow (\text{while } p \ f) : (f : x);$
 $p : x = F \rightarrow x; \perp$

The above functional forms provide an effective method for computing the values of the functions they denote (if they terminate) provided one can effectively apply their function parameters.

11.2.5 Definitions. A *definition* in an FP system is an expression of the form

Def $I \equiv r$

where the left side I is an unused function symbol and the right side r is a functional form (which may depend on I). It expresses the fact that the symbol I is to denote the function given by r . Thus the definition $\text{Def last}! \equiv ! \circ \text{reverse}$ defines the function $\text{last}!$ that produces the last element of a sequence (or \perp). Similarly,

Def $\text{last} \equiv \text{null} \circ \text{tl} \rightarrow !; \text{last} \circ \text{tl}$

defines the function last , which is the same as $\text{last}!$. Here in detail is how the definition would be used to compute $\text{last} : <1, 2>$:

$\text{last} : <1, 2> =$	$\Rightarrow (\text{null} \circ \text{tl} \rightarrow !; \text{last} \circ \text{tl}) : <1, 2>$
definition of last	$\Rightarrow \text{last} \circ \text{tl} : <1, 2>$
action of the form $(p \rightarrow f, g)$	since $\text{null} \circ \text{tl} : <1, 2> = \text{null} : <2>$
	$= F$
action of the form $f \circ g$	$\Rightarrow \text{last} : (\text{tl} : <1, 2>)$
definition of primitive tail	$\Rightarrow \text{last} : <2>$
definition of last	$\Rightarrow (\text{null} \circ \text{tl} \rightarrow !; \text{last} \circ \text{tl}) : <2>$
action of the form $(p \rightarrow f, g)$	$\Rightarrow ! : <2>$
	since $\text{null} \circ \text{tl} : <2> = \text{null} : \phi = T$
definition of selector 1	$\Rightarrow 2$

The above illustrates the simple rule: to apply a defined symbol, replace it by the right side of its definition. Of course, some definitions may define nonterminating functions. A set D of definitions is *well formed* if no two left sides are the same.

11.2.6 Semantics. It can be seen from the above that an FP system is determined by choice of the following sets: (a) The set of atoms A (which determines the set of objects). (b) The set of primitive functions P . (c) The set of functional forms F . (d) A well formed set of definitions D . To understand the semantics of such a system one needs to know how to compute $f : x$ for any function f and any object x of the system. There are exactly four possibilities for f :

- (1) f is a primitive function;
- (2) f is a functional form;
- (3) there is one definition in D , $\text{Def } f \equiv r$; and
- (4) none of the above.

If f is a primitive function, then one has its description

and knows how to apply it. If f is a functional form, then the description of the form tells how to compute $f : x$ in terms of the parameters of the form, which can be done by further use of these rules. If f is defined, $\text{Def } f \equiv r$, as in (3), then to find $f : x$ one computes $r : x$, which can be done by further use of these rules. If none of these, then $f : x = \perp$. Of course, the use of these rules may not terminate for some f and some x , in which case we assign the value $f : x = \perp$.

11.3 Examples of Functional Programs

The following examples illustrate the functional programming style. Since this style is unfamiliar to most readers, it may cause confusion at first; the important point to remember is that no part of a function definition is a result itself. Instead, each part is a *function* that must be applied to an argument to obtain a result.

11.3.1 Factorial.

Def $! \equiv \text{eq}0 \rightarrow !; \times \circ [\text{id}, ! \circ \text{sub}1]$

where

Def $\text{eq}0 \equiv \text{eq} \circ [\text{id}, 0]$

Def $\text{sub}1 \equiv - \circ [\text{id}, !]$

Here are some of the intermediate expressions an FP system would obtain in evaluating $! : 2$:

$$\begin{aligned} ! : 2 &\Rightarrow (\text{eq}0 \rightarrow !; \times \circ [\text{id}, ! \circ \text{sub}1]) : 2 \\ &\Rightarrow \times : <\text{id} : 2, ! \circ \text{sub}1 : 2> \Rightarrow \times : <2, ! : 1> \\ &\Rightarrow \times : <2, \times : <1, ! : 0>> \Rightarrow \times : <2, \times : <1, 1>> \\ &\Rightarrow \times : <2, 1> \Rightarrow 2. \end{aligned}$$

In Section 12 we shall see how theorems of the algebra of FP programs can be used to prove that $!$ is the factorial function.

11.3.2 Inner product. We have seen earlier how this definition works.

Def $\text{IP} \equiv (/+) \circ (\alpha \times) \circ \text{trans}$

11.3.3 Matrix multiply. This matrix multiplication program yields the product of any pair $<m, n>$ of conformable matrices, where each matrix m is represented as the sequence of its rows:

$m = <m_1, \dots, m_r>$

where $m_i = <m_{i1}, \dots, m_{in}>$ for $i = 1, \dots, r$.

Def $\text{MM} \equiv (\alpha \alpha \text{IP}) \circ (\alpha \text{distl}) \circ \text{distr} \circ [1, \text{trans} \circ 2]$

The program MM has four steps, reading from right to left; each is applied in turn, beginning with $[1, \text{trans} \circ 2]$, to the result of its predecessor. If the argument is $<m, n>$, then the first step yields $<m, n'>$ where $n' = \text{trans} : n$. The second step yields $<<m_1, n'>, \dots, <m_r, n'>>$, where the m_i are the rows of m . The third step, adistl , yields

$<\text{distl} : <m_1, n'>, \dots, \text{distl} : <m_r, n'>> = <p_1, \dots, p_r>$

where

$p_i = \text{distl}: \langle m_i, n' \rangle = \langle \langle m_i, n'_1 \rangle, \dots, \langle m_i, n'_r \rangle \rangle$
 for $i = 1, \dots, r$

and n'_j is the j th column of n (the j th row of n'). Thus p_i , a sequence of row and column pairs, corresponds to the i -th product row. The operator $\alpha\alpha IP$, or $\alpha(\alpha IP)$, causes αIP to be applied to each p_i , which in turn causes IP to be applied to each row and column pair in each p_i . The result of the last step is therefore the sequence of rows comprising the product matrix. If either matrix is not rectangular, or if the length of a row of m differs from that of a column of n , or if any element of m or n is not a number, the result is \perp .

This program MM does not name its arguments or any intermediate results; contains no variables, no loops, no control statements nor procedure declarations; has no initialization instructions; is not word-at-a-time in nature; is hierarchically constructed from simpler components; uses generally applicable housekeeping forms and operators (e.g., af , distl , distr , trans); is perfectly general; yields \perp whenever its argument is inappropriate in any way; does not constrain the order of evaluation unnecessarily (all applications of IP to row and column pairs can be done in parallel or in any order); and, using algebraic laws (see below), can be transformed into more "efficient" or into more "explanatory" programs (e.g., one that is recursively defined). None of these properties hold for the typical von Neumann matrix multiplication program.

Although it has an unfamiliar and hence puzzling form, the program MM describes the essential operations of matrix multiplication without overdetermining the process or obscuring parts of it, as most programs do; hence many straightforward programs for the operation can be obtained from it by formal transformations. It is an inherently inefficient program for von Neumann computers (with regard to the use of space), but efficient ones can be derived from it and realizations of FP systems can be imagined that could execute MM without the prodigal use of space it implies. Efficiency questions are beyond the scope of this paper; let me suggest only that since the language is so simple and does not dictate any binding of lambda-type variables to data, there may be better opportunities for the system to do some kind of "lazy" evaluation [9, 10] and to control data management more efficiently than is possible in lambda-calculus based systems.

11.4 Remarks About FP Systems

11.4.1 FP systems as programming languages. FP systems are so minimal that some readers may find it difficult to view them as programming languages. Viewed as such, a function f is a program, an object x is the contents of the store, and $f:x$ is the contents of the store after program f is activated with x in the store. The set of definitions is the program library. The primitive functions and the functional forms provided by the system are the basic statements of a particular programming language. Thus, depending on the choice of prim-

itive functions and functional forms, the FP framework provides for a large class of languages with various styles and capabilities. The algebra of programs associated with each of these depends on its particular set of functional forms. The primitive functions, functional forms, and programs given in this paper comprise an effort to develop just one of these possible styles.

11.4.2 Limitations of FP systems. FP systems have a number of limitations. For example, a given FP system is a fixed language; it is not history sensitive: no program can alter the library of programs. It can treat input and output only in the sense that x is an input and $f:x$ is the output. If the set of primitive functions and functional forms is weak, it may not be able to express every computable function.

An FP system cannot compute a program since function expressions are not objects. Nor can one define new functional forms within an FP system. (Both of these limitations are removed in formal functional programming (FFP) systems in which objects "represent" functions.) Thus no FP system can have a function, apply, such that

$\text{apply}: \langle x, y \rangle = x:y$

because, on the left, x is an object, and, on the right, x is a function. (Note that we have been careful to keep the set of function symbols and the set of objects distinct: thus l is a function symbol, and l is an object.)

The primary limitation of FP systems is that they are not history sensitive. Therefore they must be extended somehow before they can become practically useful. For discussion of such extensions, see the sections on FFP and AST systems (Sections 13 and 14).

11.4.3 Expressive power of FP systems. Suppose two FP systems, FP_1 and FP_2 , both have the same set of objects and the same set of primitive functions, but the set of functional forms of FP_1 properly includes that of FP_2 . Suppose also that both systems can express all computable functions on objects. Nevertheless, we can say that FP_1 is more expressive than FP_2 , since every function expression in FP_2 can be duplicated in FP_1 , but by using a functional form not belonging to FP_2 , FP_1 can express some functions more directly and easily than FP_2 .

I believe the above observation could be developed into a theory of the expressive power of languages in which a language A would be *more expressive* than language B under the following roughly stated conditions. First, form all possible functions of all types in A by applying all existing functions to objects and to each other in all possible ways until no new function of any type can be formed. (The set of objects is a type; the set of continuous functions $[T \rightarrow U]$ from type T to type U is a type. If $f \in [T \rightarrow U]$ and $t \in T$, then ft in U can be formed by applying f to t .) Do the same in language B. Next, compare each type in A to the corresponding type in B. If, for every type, A's type includes B's corresponding

type, then A is more expressive than B (or equally expressive). If some type of A's functions is incomparable to B's, then A and B are not comparable in expressive power.

11.4.4 Advantages of FP systems. The main reason FP systems are considerably simpler than either conventional languages or lambda-calculus-based languages is that they use only the most elementary fixed naming system (naming a function in a definition) with a simple fixed rule of substituting a function for its name. Thus they avoid the complexities both of the naming systems of conventional languages and of the substitution rules of the lambda calculus. FP systems permit the definition of different naming systems (see Sections 13.3.4 and 14.7) for various purposes. These need not be complex, since many programs can do without them completely. Most importantly, they treat names as functions that can be combined with other functions without special treatment.

FP systems offer an escape from conventional word-at-a-time programming to a degree greater even than APL [12] (the most successful attack on the problem to date within the von Neumann framework) because they provide a more powerful set of functional forms within a unified world of expressions. They offer the opportunity to develop higher level techniques for thinking about, manipulating, and writing programs.

12. The Algebra of Programs for FP Systems

12.1 Introduction

The algebra of the programs described below is the work of an amateur in algebra, and I want to show that it is a game amateurs can profitably play and enjoy, a game that does not require a deep understanding of logic and mathematics. In spite of its simplicity, it can help one to understand and prove things about programs in a systematic, rather mechanical way.

So far, proving a program correct requires knowledge of some moderately heavy topics in mathematics and logic: properties of complete partially ordered sets, continuous functions, least fixed points of functionals, the first-order predicate calculus, predicate transformers, weakest preconditions, to mention a few topics in a few approaches to proving programs correct. These topics have been very useful for professionals who make it their business to devise proof techniques; they have published a lot of beautiful work on this subject, starting with the work of McCarthy and Floyd, and, more recently, that of Burstall, Dijkstra, Manna and his associates, Milner, Morris, Reynolds, and many others. Much of this work is based on the foundations laid down by Dana Scott (denotational semantics) and C. A. R. Hoare (axiomatic semantics). But its theoretical level places it beyond the scope of most amateurs who work outside of this specialized field.

If the average programmer is to prove his programs

correct, he will need much simpler techniques than those the professionals have so far put forward. The algebra of programs below may be one starting point for such a proof discipline and, coupled with current work on algebraic manipulation, it may also help provide a basis for automating some of that discipline.

One advantage of this algebra over other proof techniques is that the programmer can use his programming language as the language for deriving proofs, rather than having to state proofs in a separate logical system that merely talks *about* his programs.

At the heart of the algebra of programs are laws and theorems that state that one function expression is the same as another. Thus the law $[f,g] \circ h = [f \circ h, g \circ h]$ says that the construction of f and g (composed with h) is the same function as the construction of (f composed with h) and (g composed with h) no matter what the functions f , g , and h are. Such laws are easy to understand, easy to justify, and easy and powerful to use. However, we also wish to use such laws to solve equations in which an "unknown" function appears on both sides of the equation. The problem is that if f satisfies some such equation, it will often happen that some extension f' of f will also satisfy the same equation. Thus, to give a unique meaning to solutions of such equations, we shall require a foundation for the algebra of programs (which uses Scott's notion of least fixed points of continuous functionals) to assure us that solutions obtained by algebraic manipulation are indeed least, and hence unique, solutions.

Our goal is to develop a foundation for the algebra of programs that disposes of the theoretical issues, so that a programmer can use simple algebraic laws and one or two theorems from the foundations to solve problems and create proofs in the same mechanical style we use to solve high-school algebra problems, and so that he can do so without knowing anything about least fixed points or predicate transformers.

One particular foundational problem arises: given equations of the form

$$f = p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; E_i(f), \quad (1)$$

where the p_i 's and q_i 's are functions not involving f and $E_i(f)$ is a function expression involving f , the laws of the algebra will often permit the formal "extension" of this equation by one more "clause" by deriving

$$E_i(f) = p_{i+1} \rightarrow q_{i+1}; E_{i+1}(f) \quad (2)$$

which, by replacing $E_i(f)$ in (1) by the right side of (2), yields

$$f = p_0 \rightarrow q_0; \dots; p_{i+1} \rightarrow q_{i+1}; E_{i+1}(f). \quad (3)$$

This formal extension may go on without limit. One question the foundations must then answer is: when can the least f satisfying (1) be represented by the infinite expansion

$$f = p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (4)$$

in which the final clause involving f has been dropped.

so that we now have a solution whose right side is free of f 's? Such solutions are helpful in two ways: first, they give proofs of "termination" in the sense that (4) means that $f:x$ is defined if and only if there is an n such that, for every i less than n , $p_i:x = F$ and $p_n:x = T$ and $q_n:x$ is defined. Second, (4) gives a case-by-case description of f that can often clarify its behavior.

The foundations for the algebra given in a subsequent section are a modest start toward the goal stated above. For a limited class of equations its "linear expansion theorem" gives a useful answer as to when one can go from indefinitely extendable equations like (1) to infinite expansions like (4). For a larger class of equations, a more general "expansion theorem" gives a less helpful answer to similar questions. Hopefully, more powerful theorems covering additional classes of equations can be found. But for the present, one need only know the conclusions of these two simple foundational theorems in order to follow the theorems and examples appearing in this section.

The results of the foundations subsection are summarized in a separate, earlier subsection titled "expansion theorems," without reference to fixed point concepts. The foundations subsection itself is placed later where it can be skipped by readers who do not want to go into that subject.

12.2 Some Laws of the Algebra of Programs

In the algebra of programs for an FP system variables range over the set of functions of the system. The "operations" of the algebra are the functional forms of the system. Thus, for example, $[f,g] \circ h$ is an expression of the algebra for the FP system described above, in which f , g , and h are variables denoting arbitrary functions of that system. And

$$[f,g] \circ h = [f \circ h, g \circ h]$$

is a law of the algebra which says that, whatever functions one chooses for f , g , and h , the function on the left is the same as that on the right. Thus this algebraic law is merely a restatement of the following proposition about any FP system that includes the functional forms $[f,g]$ and $f \circ g$:

PROPOSITION: For all functions f , g , and h and all objects x , $([f,g] \circ h):x = [f \circ h, g \circ h]:x$.

PROOF:

$$\begin{aligned} ([f,g] \circ h):x &= [f,g]: (h:x) \\ &= \langle f:(h:x), g:(h:x) \rangle \quad \text{by definition of composition} \\ &= \langle (f \circ h):x, (g \circ h):x \rangle \quad \text{by definition of construction} \\ &= [f \circ h, g \circ h]:x \quad \text{by definition of composition} \\ &\quad \text{by definition of construction } \square \end{aligned}$$

Some laws have a domain smaller than the domain of all objects. Thus $1 \circ [f,g] = f$ does not hold for objects x such that $g:x = \perp$. We write

$$\text{defined } g \rightarrow 1 \circ [f,g] = f$$

to indicate that the law (or theorem) on the right holds within the domain of objects x for which $\text{defined } g:x = T$. Where

$$\text{Def } \text{defined } = T$$

i.e. $\text{defined } : x = x = \perp \rightarrow \perp ; T$. In general we shall write a *qualified functional equation*:

$$p \rightarrow f = g$$

to mean that, for any object x , whenever $p:x = T$, then $f:x = g:x$.

Ordinary algebra concerns itself with two operations, addition and multiplication; it needs few laws. The algebra of programs is concerned with more operations (functional forms) and therefore needs more laws.

Each of the following laws requires a corresponding proposition to validate it. The interested reader will find most proofs of such propositions easy (two are given below). We first define the usual ordering on functions and equivalence in terms of this ordering:

DEFINITION $f \leq g$ iff for all objects x , either $f:x = \perp$, or $f:x = g:x$.

DEFINITION $f = g$ iff $f \leq g$ and $g \leq f$.

It is easy to verify that \leq is a partial ordering, that $f \leq g$ means g is an extension of f , and that $f = g$ iff $f:x = g:x$ for all objects x . We now give a list of algebraic laws organized by the two principal functional forms involved.

I Composition and construction

- I.1 $[f_1, \dots, f_n] \circ g = [f_1 \circ g, \dots, f_n \circ g]$
- I.2 $\alpha f \circ [g_1, \dots, g_n] = [f \circ g_1, \dots, f \circ g_n]$
- I.3 $/f \circ [g_1, \dots, g_n]$
 - $= f \circ [g_1, /f \circ [g_2, \dots, g_n]] \quad \text{when } n \geq 2$
 - $= f \circ [g_1, f \circ [g_2, \dots, f \circ [g_{n-1}, g_n] \dots]]$
- I.4 $/f \circ [\bar{x}, g] = (\bar{b} u f x) \circ g$
- I.5 $1 \circ [f_1, \dots, f_n] \leq f_1$
 - $s \circ [f_1, \dots, f_n] \leq f_s \text{ for any selector } s, s \leq n$
 - $\text{defined } f_i \text{ (for all } i \neq s, 1 \leq i \leq n\text{)} \rightarrow \rightarrow$
 - $s \circ [f_1, \dots, f_n] = f_s$
- I.5.1 $[f_1 \circ 1, \dots, f_n \circ n] \circ [g_1, \dots, g_n] = [f_1 \circ g_1, \dots, f_n \circ g_n]$
- I.6 $\text{tl} \circ [f_1] \leq \phi \text{ and}$
 - $\text{tl} \circ [f_1, \dots, f_n] \leq [f_2, \dots, f_n] \quad \text{for } n \geq 2$
 - $\text{defined } f_1 \rightarrow \rightarrow \text{tl} \circ [f_1] = \phi$
 - $\text{and } \text{tl} \circ [f_1, \dots, f_n] = [f_2, \dots, f_n] \text{ for } n \geq 2$
- I.7 $\text{distl} \circ [f, [g_1, \dots, g_n]] = [[f, g_1], \dots, [f, g_n]]$
 - $\text{defined } f \rightarrow \rightarrow \text{distl} \circ [f, \phi] = \phi$
 - The analogous law holds for distr.
- I.8 $\text{apndl} \circ [f, [g_1, \dots, g_n]] = [f, g_1, \dots, g_n]$
 - $\text{null} \circ g \rightarrow \rightarrow \text{apndl} \circ [f, g] = [f]$
- I.9 And so on for apndr, reverse, rotl, etc.
- I.10 $\text{apndl} \circ [f \circ g, \alpha f \circ h] = \alpha f \circ \text{apndl} \circ [g, h]$
- I.11 $\text{pair} \& \text{not} \circ \text{null} \circ 1 \rightarrow \rightarrow$
 - $\text{apndl} \circ [[1 \circ 1, 2], \text{distr} \circ [\text{tl} \circ 1, 2]] = \text{distr}$

Where $f \& g = \text{and} \circ [f, g]$;
 $\text{pair} = \text{atom} \rightarrow \bar{F}; \text{eq} \circ [\text{length}, 2]$

II Composition and condition (right associated parentheses omitted) (Law II.2 is noted in Manna et al. [16], p. 493.)

$$\begin{aligned} \text{II.1 } & (p \rightarrow f; g) \circ h \equiv p \circ h \rightarrow f \circ h; g \circ h \\ \text{II.2 } & h \circ (p \rightarrow f; g) \equiv p \rightarrow h \circ f, h \circ g \\ \text{II.3 } & \text{or} \circ [q, \text{not} \circ q] \rightarrow \rightarrow \text{and} \circ [p, q] \rightarrow f; \\ & \quad \text{and} \circ [p, \text{not} \circ q] \rightarrow g; h \equiv p \rightarrow (q \rightarrow f; g); h \end{aligned}$$

$$\text{II.3.1 } p \rightarrow (p \rightarrow f; g); h \equiv p \rightarrow f; h$$

III Composition and miscellaneous

$$\text{III.1 } \bar{x} \circ f \leq \bar{x}$$

defined of $\rightarrow \bar{x} \circ f \equiv \bar{x}$

$$\text{III.1.1 } \bar{1} \circ f \equiv f \circ \bar{1} \equiv \bar{1}$$

$$\text{III.2 } f \circ \text{id} \equiv \text{id} \circ f \equiv f$$

$$\text{III.3 } \text{pair} \rightarrow \rightarrow \text{l} \circ \text{distr} \equiv [\text{l} \circ \text{l}, 2] \text{ also:}$$

$$\text{pair} \rightarrow \rightarrow \text{l} \circ \text{tl} \equiv 2 \text{ etc.}$$

$$\text{III.4 } \alpha(f \circ g) \equiv \alpha f \circ \alpha g$$

$$\text{III.5 } \text{null} \circ g \rightarrow \rightarrow \alpha f \circ g \equiv \bar{\phi}$$

IV Condition and construction

$$\text{IV.1 } [f_1, \dots, (p \rightarrow g, h), \dots, f_n]$$

$$= p \rightarrow [f_1, \dots, g, \dots, f_n]; [f_1, \dots, h, \dots, f_n]$$

$$\text{IV.1.1 } [f_1, \dots, (p_1 \rightarrow g_1; \dots; p_n \rightarrow g_n, h), \dots, f_m]$$

$$= p_1 \rightarrow [f_1, \dots, g_1, \dots, f_m];$$

$$\dots; p_n \rightarrow [f_1, \dots, g_n, \dots, f_m]; [f_1, \dots, h, \dots, f_m]$$

This concludes the present list of algebraic laws; it is by no means exhaustive, there are many others.

Proof of two laws

We give the proofs of validating propositions for laws I.10 and I.11, which are slightly more involved than most of the others.

PROPOSITION 1

$$\text{apndl} \circ [f \circ g, \alpha f \circ h] \equiv \alpha f \circ \text{apndl} \circ [g, h]$$

PROOF. We show that, for every object x , both of the above functions yield the same result.

CASE 1. $h : x$ is neither a sequence nor $\bar{\phi}$.

Then both sides yield \perp when applied to x .

CASE 2. $h : x = \bar{\phi}$. Then

$$\text{apndl} \circ [f \circ g, \alpha f \circ h] : x$$

$$= \text{apndl}: \langle f \circ g : x, \bar{\phi} \rangle = \langle f : (g : x) \rangle$$

$$\alpha f \circ \text{apndl} \circ [g, h] : x$$

$$= \alpha f \circ \text{apndl}: \langle g : x, \bar{\phi} \rangle = \alpha f: \langle g : x \rangle$$

$$= \langle f : (g : x) \rangle$$

CASE 3. $h : x = \langle y_1, \dots, y_n \rangle$. Then

$$\text{apndl} \circ [f \circ g, \alpha f \circ h] : x$$

$$= \text{apndl}: \langle f \circ g : x, \alpha f: \langle y_1, \dots, y_n \rangle \rangle$$

$$= \langle f : (g : x), f : y_1, \dots, f : y_n \rangle$$

$$\alpha f \circ \text{apndl} \circ [g, h] : x$$

$$= \alpha f \circ \text{apndl}: \langle g : x, \langle y_1, \dots, y_n \rangle \rangle$$

$$= \alpha f: \langle g : x, y_1, \dots, y_n \rangle$$

$$= \langle f : (g : x), f : y_1, \dots, f : y_n \rangle$$

PROPOSITION 2

$$\text{Pair} \& \text{not} \circ \text{null} \circ \text{l} \rightarrow \rightarrow$$

$$\text{apndl} \circ [[1^2, 2], \text{distr} \circ [\text{tl} \circ \text{l}, 2]] \equiv \text{distr}$$

where $f \& g$ is the function: $\text{and} \circ [f, g]$, and $f^2 \equiv f \circ f$.

PROOF. We show that both sides produce the same result when applied to any pair $\langle x, y \rangle$, where $x \neq \bar{\phi}$, as per the stated qualification.

CASE 1. x is an atom or \perp . Then $\text{distr}: \langle x, y \rangle = \perp$, since $x \neq \bar{\phi}$. The left side also yields \perp when applied to $\langle x, y \rangle$, since $\text{tl} \circ \text{l}: \langle x, y \rangle = \perp$ and all functions are \perp -preserving.

CASE 2. $x = \langle x_1, \dots, x_n \rangle$. Then

$$\text{apndl} \circ [[1^2, 2], \text{distr} \circ [\text{tl} \circ \text{l}, 2]]: \langle x, y \rangle$$

$$= \text{apndl}: \langle \langle \text{tl}: x, y \rangle, \text{distr}: \langle \text{tl}: x, y \rangle \rangle$$

$$= \text{apndl}: \langle \langle \text{tl}: x, y \rangle, \phi \rangle = \langle \langle \text{tl}: x, y \rangle \rangle \text{ if tl}: x = \phi$$

$$= \text{apndl}: \langle \langle \text{tl}: x, y \rangle, \langle \langle \text{tl}: x, y \rangle, \dots, \langle \text{tl}: x, y \rangle \rangle \rangle$$

$$\text{if tl}: x \neq \phi$$

$$= \langle \langle x_1, y \rangle, \dots, \langle x_n, y \rangle \rangle$$

$$= \text{distr}: \langle x, y \rangle$$

□

12.3 Example: Equivalence of Two Matrix Multiplication Programs

We have seen earlier the matrix multiplication program:

$$\text{Def MM} \equiv \alpha \alpha \text{IP} \circ \alpha \text{distl} \circ \text{distr} \circ [1, \text{trans} \circ 2].$$

We shall now show that its initial segment, MM' , where

$$\text{Def MM}' \equiv \alpha \alpha \text{IP} \circ \alpha \text{distl} \circ \text{distr},$$

can be defined recursively. (MM' “multiplies” a pair of matrices after the second matrix has been transposed. Note that MM' , unlike MM , gives \perp for all arguments that are not pairs.) That is, we shall show that MM' satisfies the following equation which recursively defines the same function (on pairs):

$$f \equiv \text{null} \circ \text{l} \rightarrow \bar{\phi}; \text{apndl} \circ [\alpha \text{IP} \circ \text{distl} \circ [1 \circ 1, 2], f \circ [\text{tl} \circ \text{l}, 2]].$$

Our proof will take the form of showing that the following function, R ,

$$\text{Def R} \equiv \text{null} \circ \text{l} \rightarrow \bar{\phi};$$

$$\text{apndl} \circ [\alpha \text{IP} \circ \text{distl} \circ [1 \circ 1, 2], \text{MM}' \circ [\text{tl} \circ \text{l}, 2]]$$

is, for all pairs $\langle x, y \rangle$, the same function as MM' . R “multiplies” two matrices, when the first has more than zero rows, by computing the first row of the “product” (with $\alpha \text{IP} \circ \text{distl} \circ [1 \circ 1, 2]$) and adjoining it to the “product” of the tail of the first matrix and the second matrix. Thus the theorem we want is

$$\text{pair} \rightarrow \rightarrow \text{MM}' \equiv R,$$

from which the following is immediate:

$$\text{MM} \equiv \text{MM}' \circ [1, \text{trans} \circ 2] \equiv R \circ [1, \text{trans} \circ 2];$$

where

$$\text{Def pair} \equiv \text{atom} \rightarrow \bar{F}; \text{eq} \circ [\text{length}, 2].$$

$$\text{THEOREM: pair} \rightarrow \rightarrow \text{MM}' \equiv R$$

where

Def $MM' \equiv \alpha\alpha IP \circ \alpha distl \circ distr$

Def $R \equiv \text{null} \circ 1 \rightarrow \bar{\phi}$;

$$\text{apndl} \circ [\alpha IP \circ distl \circ [1^2, 2], MM' \circ [tl \circ 1, 2]]$$

PROOF.

CASE 1. pair & null $\circ 1 \rightarrow MM' = R$.

pair & null $\circ 1 \rightarrow R = \bar{\phi}$ by def of R

pair & null $\circ 1 \rightarrow MM' = \bar{\phi}$

since distr: $\langle \bar{\phi}, x \rangle = \bar{\phi}$ by def of distr

and $\alpha f \circ \bar{\phi} = \bar{\phi}$ by def of Apply to all.

And so: $\alpha\alpha IP \circ \alpha distl \circ distr: \langle \bar{\phi}, x \rangle = \bar{\phi}$.

Thus pair & null $\circ 1 \rightarrow MM' = R$.

CASE 2. pair & not \circ null $\circ 1 \rightarrow MM' = R$.

pair & not \circ null $\circ 1 \rightarrow R = R'$,

(1)

by def of R and R' , where

Def $R' \equiv \text{apndl} \circ [\alpha IP \circ distl \circ [1^2, 2], MM' \circ [tl \circ 1, 2]]$.

We note that

$$R' \equiv \text{apndl} \circ [f \circ g, \alpha f \circ h]$$

where

$$f \equiv \alpha IP \circ distl$$

$$g \equiv [1^2, 2]$$

$$h \equiv \text{distr} \circ [tl \circ 1, 2]$$

$$\alpha f \equiv \alpha(\alpha IP \circ distl) = \alpha\alpha IP \circ \alpha distl \quad (\text{by III.4}). \quad (2)$$

Thus, by I.10,

$$R' \equiv \alpha f \circ \text{apndl} \circ [g, h]. \quad (3)$$

Now $\text{apndl} \circ [g, h] \equiv \text{apndl} \circ [[1^2, 2], \text{distr} \circ [tl \circ 1, 2]]$,
thus, by I.11,

$$\text{pair} \& \text{not} \circ \text{null} \circ 1 \rightarrow \text{apndl} \circ [g, h] = \text{distr}. \quad (4)$$

And so we have, by (1), (2), (3) and (4),

$$\begin{aligned} \text{pair} \& \text{not} \circ \text{null} \circ 1 \rightarrow R &= R' \\ &\equiv \alpha f \circ \text{distr} \equiv \alpha\alpha IP \circ \alpha distl \circ \text{distr} \equiv MM'. \end{aligned}$$

Case 1 and Case 2 together prove the theorem. \square

12.4 Expansion Theorems

In the following subsections we shall be “solving” some simple equations (where by a “solution” we shall mean the “least” function which satisfies an equation). To do so we shall need the following notions and results drawn from the later subsection on foundations of the algebra, where their proofs appear.

12.4.1 Expansion. Suppose we have an equation of the form

$$f \equiv E(f) \quad (E1)$$

where $E(f)$ is an expression involving f . Suppose further that there is an infinite sequence of functions f_i for $i = 0, 1, 2, \dots$, each having the following form:

$$\begin{aligned} f_0 &\equiv \perp \\ f_{i+1} &\equiv p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; \perp \end{aligned} \quad (E2)$$

where the p_i 's and q_i 's are particular functions, so that E has the property:

$$E(f_i) = f_{i+1} \text{ for } i = 0, 1, 2, \dots \quad (E3)$$

Then we say that E is *expansive* and has the f_i 's as *approximating functions*.

If E is expansive and has approximating functions as in (E2), and if f is the solution of (E1), then f can be written as the infinite expansion

$$f = p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (E4)$$

meaning that, for any x , $f.x \neq \perp$ iff there is an $n \geq 0$ such that (a) $p_i.x = F$ for all $i < n$, and (b) $p_n.x = T$, and (c) $q_n.x \neq \perp$. When $f.x \neq \perp$, then $f.x = q_n.x$ for this n . (The foregoing is a consequence of the “expansion theorem”.)

12.4.2 Linear expansion. A more helpful tool for solving some equations applies when, for any function h ,

$$E(h) \equiv p_0 \rightarrow q_0; E_1(h) \quad (LE1)$$

and there exist p_i and q_i such that

$$E_1(p_i \rightarrow q_i; h) = p_{i+1} \rightarrow q_{i+1}; E_1(h) \quad \text{for } i = 0, 1, 2, \dots \quad (LE2)$$

and

$$E_1(\perp) = \perp. \quad (LE3)$$

Under the above conditions E is said to be *linearly expansive*. If so, and f is the solution of

$$f \equiv E(f) \quad (LE4)$$

then E is expansive and f can again be written as the infinite expansion

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (LE5)$$

using the p_i 's and q_i 's generated by (LE1) and (LE2).

Although the p_i 's and q_i 's of (E4) or (LE5) are not unique for a given function, it may be possible to find additional constraints which would make them so, in which case the expansion (LE5) would comprise a canonical form for a function. Even without uniqueness these expansions often permit one to prove the equivalence of two different function expressions, and they often clarify a function's behavior.

12.5 A Recursion Theorem

Using three of the above laws and linear expansion, one can prove the following theorem of moderate generality that gives a clarifying expansion for many recursively defined functions.

RECURSION THEOREM: Let f be a solution of

$$f = p \rightarrow g; Q(f) \quad (1)$$

where

$$Q(k) = h \circ [i, k \circ j] \text{ for any function } k \quad (2)$$

and p, g, h, i, j are any given functions, then

$$f = p \rightarrow g; p \circ j \rightarrow Q(g); \dots ; p \circ j^n \rightarrow Q^n(g); \dots \quad (3)$$

(where $Q^n(g)$ is $h \circ [i, Q^{n-1}(g) \circ j]$, and j^n is $j \circ j^{n-1}$ for $n \geq 2$) and

$$Q^n(g) = /h \circ [i, i \circ j, \dots, i \circ j^{n-1}, g \circ j^n]. \quad (4)$$

PROOF. We verify that $p \rightarrow g; Q(f)$ is linearly expansive. Let p_n, q_n and k be any functions. Then

$$\begin{aligned} Q(p_n \rightarrow q_n; k) &= h \circ [i, (p_n \rightarrow q_n; k) \circ j] \quad \text{by (2)} \\ &= h \circ [i, (p_n \circ j \rightarrow q_n \circ j; k \circ j)] \quad \text{by II.1} \\ &= h \circ (p_n \circ j \rightarrow [i, q_n \circ j]; [i, k \circ j]) \quad \text{by IV.1} \\ &\equiv p_n \circ j \rightarrow h \circ [i, q_n \circ j]; h \circ [i, k \circ j] \quad \text{by II.2} \\ &\equiv p_n \circ j \rightarrow Q(q_n); Q(k) \quad \text{by (2)} \end{aligned} \quad (5)$$

Thus if $p_0 = p$ and $q_0 = g$, then (5) gives $p_1 = p \circ j$ and $q_1 = Q(g)$ and in general gives the following functions satisfying (LE2)

$$p_n = p \circ j^n \quad \text{and} \quad q_n = Q^n(g). \quad (6)$$

Finally,

$$\begin{aligned} Q(\bar{I}) &\equiv h \circ [i, \bar{I} \circ j] \\ &\equiv h \circ [i, \bar{I}] \quad \text{by III.1.1} \\ &\equiv h \circ \bar{I} \quad \text{by I.9} \\ &\equiv \bar{I} \quad \text{by III.1.1.} \end{aligned} \quad (7)$$

Thus (5) and (6) verify (LE2) and (7) verifies (LE3), with $E_1 \equiv Q$. If we let $E(f) \equiv p \rightarrow g; Q(f)$, then we have (LE1); thus E is linearly expansive. Since f is a solution of $f = E(f)$, conclusion (3) follows from (6) and (LE5). Now

$$\begin{aligned} Q^n(g) &\equiv h \circ [i, Q^{n-1}(g) \circ j] \\ &\equiv h \circ [i, h \circ [i \circ j, \dots, h \circ [i \circ j^{n-1}, g \circ j^n] \dots]] \\ &\quad \text{by I.1, repeatedly} \\ &\equiv /h \circ [i, i \circ j, \dots, i \circ j^{n-1}, g \circ j^n] \quad \text{by I.3} \end{aligned} \quad (8)$$

Result (8) is the second conclusion (4). \square

12.5.1 Example: correctness proof of a recursive factorial function. Let f be a solution of

$$f = \text{eq0} \rightarrow \bar{I}; \times \circ [\text{id}, f \circ s]$$

where

$$\text{Def } s = -\circ [\text{id}, \bar{I}] \quad (\text{subtract 1}).$$

Then f satisfies the hypothesis of the recursion theorem with $p = \text{eq0}$, $g = \bar{I}$, $h = \times$, $i = \text{id}$, and $j = s$. Therefore

$$f = \text{eq0} \rightarrow \bar{I}; \dots; \text{eq0} \circ s^n \rightarrow Q^n(\bar{I}); \dots$$

and

$$Q^n(\bar{I}) = / \times \circ [\text{id}, \text{id} \circ s, \dots, \text{id} \circ s^{n-1}, \bar{I} \circ s^n].$$

Now $\text{id} \circ s^k \equiv s^k$ by III.2 and $\text{eq0} \circ s^n \rightarrow \bar{I} \circ s^n \equiv \bar{I}$ by III.1, since $\text{eq0} \circ s^n : x$ implies $\text{defined} \circ s^n : x$; and also $\text{eq0} \circ s^n : x \equiv \text{eq0} : (x - n) \equiv x = n$. Thus if $\text{eq0} \circ s^n : x = T$, then $x = n$ and

$$\begin{aligned} Q^n(\bar{I}) : n &= n \times (n - 1) \times \dots \times (n - (n - 1)) \\ &\quad \times (\bar{I} : (n - n)) = n!. \end{aligned}$$

Using these results for $\bar{I} \circ s^n$, $\text{eq0} \circ s^n$, and $Q^n(\bar{I})$ in the previous expansion for f , we obtain

$$\begin{aligned} f : x = 0 &\rightarrow \bar{I}; \dots; x = n \\ &\rightarrow n \times (n - 1) \times \dots \times 1 \times 1; \dots \end{aligned}$$

Thus we have proved that f terminates on precisely the set of nonnegative integers and that it is the factorial function thereon.

12.6 An Iteration Theorem

This is really a corollary of the recursion theorem. It gives a simple expansion for many iterative programs.

ITERATION THEOREM: Let f be the solution (i.e., the least solution) of

$$f = p \rightarrow g; h \circ f \circ k$$

then

$$f = p \rightarrow g; p \circ k \rightarrow h \circ g \circ k; \dots; p \circ k^n \rightarrow h^n \circ g \circ k^n; \dots$$

PROOF. Let $h' \equiv h \circ 2$, $i' \equiv \text{id}$, $f' \equiv k$, then

$$f \equiv p \rightarrow g; h' \circ [i', f' \circ f']$$

since $h \circ 2 \circ [\text{id}, f' \circ f] \equiv h \circ f' \circ f$ by I.5 (id is defined except for \perp , and the equation holds for \perp). Thus the recursion theorem gives

$$f = p \rightarrow g; \dots; p \circ k^n \rightarrow Q^n(g); \dots$$

where

$$Q^n(g) \equiv h \circ 2 \circ [\text{id}, Q^{n-1}(g) \circ k]$$

$$\equiv h \circ Q^{n-1}(g) \circ k \equiv h^n \circ g \circ k^n$$

by I.5 \square

12.6.1 Example: Correctness proof for an iterative factorial function. Let f be the solution of

$$f = \text{eq0} \circ 1 \rightarrow 2; f \circ [s \circ 1, \times]$$

where **Def** $s = -\circ [\text{id}, \bar{I}]$ (subtract 1). We want to prove that $f : \langle x, I \rangle = x!$ iff x is a nonnegative integer. Let $p = \text{eq0} \circ 1$, $g = 2$, $h = \text{id}$, $i = [s \circ 1, \times]$. Then

$$f = p \rightarrow g; h \circ f \circ k$$

and so

$$f = p \rightarrow g; \dots; p \circ k^n \rightarrow g \circ k^n; \dots \quad (1)$$

by the iteration theorem, since $h^n \equiv \text{id}$. We want to show that

$$\text{pair} \rightarrow \rightarrow k^n \equiv [a_n, b_n] \quad (2)$$

holds for every $n \geq 1$, where

$$a_n \equiv s^n \circ \{ \quad (3)$$

$$b_n \equiv / \times \circ [s^{n-1} \circ 1, \dots, s \circ 1, 1, 2] \quad (4)$$

Now (2) holds for $n = 1$ by definition of k . We assume it holds for some $n \geq 1$ and prove it then holds for $n + 1$. Now

$$\text{pair} \rightarrow \rightarrow k^{n+1} \equiv k \circ k^n \equiv [s \circ 1, \times] \circ [a_n, b_n] \quad (5)$$

since (2) holds for n . And so

pair $\rightarrow k^{n+1} \equiv [s \circ a_n, \times \circ [a_n, b_n]]$ by I.1 and I.5 (6)

To pass from (5) to (6) we must check that whenever a_n or b_n yield \perp in (5), so will the right side of (6). Now

$$s \circ a_n \equiv s^{n+1} \circ 1 = a_{n+1} \quad (7)$$

$$\begin{aligned} \times \circ [a_n, b_n] &\equiv / \times \circ [s^n \circ 1, s^{n-1} \circ 1, \dots, s \circ 1, 1, 2] \\ &= b_{n+1}, \text{ by I.3.} \end{aligned} \quad (8)$$

Combining (6), (7), and (8) gives

$$\text{pair } \rightarrow k^{n+1} \equiv [a_{n+1}, b_{n+1}]. \quad (9)$$

Thus (2) holds for $n = 1$ and holds for $n + 1$ whenever it holds for n , therefore, by induction, it holds for every $n \geq 1$. Now (2) gives, for pairs:

$$\begin{aligned} \text{defined} \circ k^n \rightarrow p \circ k^n &\equiv \text{eq}0 \circ 1 \circ [a_n, b_n] \\ &\equiv \text{eq}0 \circ a_n \equiv \text{eq}0 \circ s^n \circ 1 \end{aligned} \quad (10)$$

$$\begin{aligned} \text{defined} \circ k^n \rightarrow g \circ k^n &= 2 \circ [a_n, b_n] \equiv / \times \circ [s^{n-1} \circ 1, \dots, s \circ 1, 1, 2] \end{aligned} \quad (11)$$

(both use I.5). Now (1) tells us that $f: \langle x, I \rangle$ is defined iff there is an n such that $p \circ k^i: \langle x, I \rangle = F$ for all $i < n$, and $p \circ k^n: \langle x, I \rangle = T$, that is, by (10), $\text{eq}0 \circ s^n \circ x = T$, i.e., $x = n$; and $g \circ k^n: \langle x, I \rangle$ is defined, in which case, by (11),

$$f: \langle x, I \rangle = / \times: \langle 1, 2, \dots, x-1, x, I \rangle = n!,$$

which is what we set out to prove.

12.6.2 Example: proof of equivalence of two iterative programs. In this example we want to prove that two iteratively defined programs, f and g , are the same function. Let f be the solution of

$$f \equiv p \circ 1 \rightarrow 2; h \circ f \circ [k \circ 1, 2]. \quad (1)$$

Let g be the solution of

$$g \equiv p \circ 1 \rightarrow 2; g \circ [k \circ 1, h \circ 2]. \quad (2)$$

Then, by the iteration theorem:

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (3)$$

$$g \equiv p'_0 \rightarrow q'_0; \dots; p'_n \rightarrow q'_n; \dots \quad (4)$$

where (letting $r^0 = \text{id}$ for any r), for $n = 0, 1, \dots$

$$p_n \equiv p \circ 1 \circ [k \circ 1, 2]^n \equiv p \circ 1 \circ [k^n \circ 1, 2] \quad \text{by I.5.1} \quad (5)$$

$$q_n \equiv h^n \circ 2 \circ [k \circ 1, 2]^n \equiv h^n \circ 2 \circ [k^n \circ 1, 2] \quad \text{by I.5.1} \quad (6)$$

$$p'_n \equiv p \circ 1 \circ [k \circ 1, h \circ 2]^n \equiv p \circ 1 \circ [k^n \circ 1, h^n \circ 2] \quad \text{by I.5.1} \quad (7)$$

$$q'_n \equiv 2 \circ [k \circ 1, h \circ 2]^n \equiv 2 \circ [k^n \circ 1, h^n \circ 2] \quad \text{by I.5.1.} \quad (8)$$

Now, from the above, using I.5,

$$\text{defined} \circ 2 \rightarrow p_n \equiv p \circ k^n \circ 1 \quad (9)$$

$$\text{defined} \circ h^n \circ 2 \rightarrow p'_n \equiv p \circ k^n \circ 1 \quad (10)$$

$$\text{defined} \circ k^n \circ 1 \rightarrow q_n \equiv q'_n \equiv h^n \circ 2 \quad (11)$$

Thus

$$\text{defined} \circ h^n \circ 2 \rightarrow \text{defined} \circ 2 = T \quad (12)$$

$$\text{defined} \circ h^n \circ 2, \rightarrow p_n \equiv p'_n \quad (13)$$

and

$$f \equiv p_0 \rightarrow q_0; \dots; p_n \rightarrow h^n \circ 2; \dots \quad (14)$$

$$g \equiv p'_0 \rightarrow q'_0; \dots; p'_n \rightarrow h^n \circ 2; \dots \quad (15)$$

since p_n and p'_n provide the qualification needed for $q_n = q'_n \equiv h^n \circ 2$.

Now suppose there is an x such that $f: x \neq g: x$. Then there is an n such that $p_i: x = p'_i: x = F$ for $i < n$, and $p_n: x \neq p'_n: x$. From (12) and (13) this can only happen when $h^n \circ 2: x = \perp$. But since h is \perp -preserving, $h^m \circ 2: x = \perp$ for all $m \geq n$. Hence $f: x = g: x = \perp$ by (14) and (15). This contradicts the assumption that there is an x for which $f: x \neq g: x$. Hence $f \equiv g$.

This example (by J. H. Morris, Jr.) is treated more elegantly in [16] on p. 498. However, some may find that the above treatment is more constructive, leads one more mechanically to the key questions, and provides more insight into the behavior of the two functions.

12.7 Nonlinear Equations

The preceding examples have concerned “linear” equations (in which the “unknown” function does not have an argument involving itself). The question of the existence of simple expansions that “solve” “quadratic” and higher order equations remains open.

The earlier examples concerned solutions of $f \equiv E(f)$, where E is linearly expansive. The following example involves an $E(f)$ that is quadratic and expansive (but not linearly expansive).

12.7.1 Example: proof of idempotency ([16] p. 497). Let f be the solution of

$$f \equiv E(f) \equiv p \rightarrow \text{id}; f^2 \circ h. \quad (1)$$

We wish to prove that $f = f^2$. We verify that E is expansive (Section 12.4.1) with the following approximating functions:

$$f_0 \equiv \perp \quad (2a)$$

$$f_n \equiv p \rightarrow \text{id}; \dots; p \circ h^{n-1} \rightarrow h^{n-1}; \perp \quad \text{for } n > 0 \quad (2b)$$

First we note that $p \rightarrow f_n = \text{id}$ and so

$$p \circ h^i \rightarrow f_n \circ h^i \equiv h^i. \quad (3)$$

$$\text{Now } E(f_0) \equiv p \rightarrow \text{id}; \perp^2 \circ h \equiv f_1, \quad (4)$$

and

$$\begin{aligned} E(f_n) &\equiv p \rightarrow \text{id}; f_n \circ (p \rightarrow \text{id}; \dots; p \circ h^{n-1} \rightarrow h^{n-1}; \perp) \circ h \\ &\equiv p \rightarrow \text{id}; f_n \circ (p \circ h \rightarrow h; \dots; p \circ h^n \rightarrow h^n; \perp \circ h) \\ &\equiv p \rightarrow \text{id}; p \circ h \rightarrow f_n \circ h; \dots; p \circ h^n \rightarrow f_n \circ h^n; f_n \circ \perp \\ &\equiv p \rightarrow \text{id}; p \circ h \rightarrow h; \dots; p \circ h^n \rightarrow h^n; \perp \quad \text{by (3)} \\ &\equiv f_{n+1}. \end{aligned} \quad (5)$$

Thus E is expansive by (4) and (5); so by (2) and Section 12.4.1 (E4)

$$f \equiv p \rightarrow \text{id}; \dots; p \circ h^n \rightarrow h^n; \dots \quad (6)$$

But (6), by the iteration theorem, gives

$$f \equiv p \rightarrow \text{id}; f \circ h. \quad (7)$$

Now, if $p: x = T$, then $f: x = x = f^2 \circ h: x$, by (1). If $p: x = F$, then

$$f: x = f^2 \circ h: x \quad \text{by (1)}$$

$$= f(f \circ h; x) = f(f;x) \text{ by (7)} \\ = f^2;x.$$

If $p;x$ is neither T nor F , then $f;x = \perp = f^2;x$. Thus
 $f = f^2$.

12.8 Foundations for the Algebra of Programs

Our purpose in this section is to establish the validity of the results stated in Section 12.4. Subsequent sections do not depend on this one, hence it can be skipped by readers who wish to do so. We use the standard concepts and results from [16], but the notation used for objects and functions, etc., will be that of this paper.

We take as the domain (and range) for all functions the set O of objects (which includes \perp) of a given FP system. We take F to be the set of functions, and \mathbf{F} to be the set of functional forms of that FP system. We write $E(f)$ for any function expression involving functional forms, primitive and defined functions, and the function symbol f ; and we regard E as a functional that maps a function f into the corresponding function $E(f)$. We assume that all $f \in F$ are \perp -preserving and that all functional forms in \mathbf{F} correspond to continuous functionals in every variable (e.g., $[f, g]$ is continuous in both f and g). (All primitive functions of the FP system given earlier are \perp -preserving, and all its functional forms are continuous.)

DEFINITIONS. Let $E(f)$ be a function expression. Let

$$f_0 = \perp$$

$$f_{i+1} = p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; \perp \text{ for } i = 0, 1, \dots$$

where $p_i, q_i \in F$. Let E have the property that

$$E(f_i) = f_{i+1} \text{ for } i = 0, 1, \dots$$

Then E is said to be *expansive* with the *approximating functions* f_i . We write

$$f = p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots$$

to mean that $f = \lim_i \{f_i\}$, where the f_i have the form above. We call the right side an *infinite expansion* of f . We take $f;x$ to be defined iff there is an $n \geq 0$ such that (a) $p_i;x = F$ for all $i < n$, and (b) $p_n;x = T$, and (c) $q_n;x$ is defined, in which case $f;x = q_n;x$.

EXPANSION THEOREM: Let $E(f)$ be expansive with approximating functions as above. Let f be the least function satisfying

$$f = E(f).$$

Then

$$f = p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots$$

PROOF. Since E is the composition of continuous functionals (from \mathbf{F}) involving only monotonic functions (\perp -preserving functions from F) as constant terms, E is continuous ([16] p. 493). Therefore its least fixed point f is $\lim_i \{E^i(\perp)\} = \lim_i \{f_i\}$ ([16] p. 494), which by definition is the above infinite expansion for f . \square

DEFINITION. Let $E(f)$ be a function expression satisfying the following:

$$E(h) = p_i \rightarrow q_i; E_1(h) \text{ for all } h \in F \quad (\text{LE1})$$

where $p_i \in F$ and $q_i \in F$ exist such that

$$E_1(p_i \rightarrow q_i; h) = p_{i+1} \rightarrow q_{i+1}; E_1(h) \text{ for all } h \in F \text{ and } i = 0, 1, \dots \quad (\text{LE2})$$

and

$$E_1(\perp) = \perp. \quad (\text{LE3})$$

Then E is said to be *linearly expansive* with respect to these p_i 's and q_i 's.

LINEAR EXPANSION THEOREM: Let E be linearly expansive with respect to p_i and q_i , $i = 0, 1, \dots$. Then E is expansive with approximating functions

$$f_0 = \perp \quad (1)$$

$$f_{i+1} = p_0 \rightarrow q_0; \dots; p_i \rightarrow q_i; \perp. \quad (2)$$

PROOF. We want to show that $E(f_i) = f_{i+1}$ for any $i \geq 0$. Now

$$E(f_0) = p_0 \rightarrow q_0; E_1(\perp) = p_0 \rightarrow q_0; \perp = f_1 \quad (3)$$

by (LE1) (LE3) (1).

Let $i > 0$ be fixed and let

$$f_i = p_0 \rightarrow q_0; w_1 \quad (4a)$$

$$w_1 = p_1 \rightarrow q_1; w_2 \quad (4b)$$

etc.

$$w_{i-1} = p_{i-1} \rightarrow q_{i-1}; \perp. \quad (4)$$

Then, for this $i > 0$

$$E(f_i) = p_0 \rightarrow q_0; E_1(f_i) \text{ by (LE1)}$$

$$E_1(f_i) = p_1 \rightarrow q_1; E_1(w_1) \text{ by (LE2) and (4a)}$$

$$E_1(w_1) = p_2 \rightarrow q_2; E_1(w_2) \text{ by (LE2) and (4b)}$$

etc.

$$E_1(w_{i-1}) = p_i \rightarrow q_i; E_1(\perp) \text{ by (LE2) and (4)} \\ = p_i \rightarrow q_i; \perp \text{ by (LE3)}$$

Combining the above gives

$$E(f_i) = f_{i+1} \text{ for arbitrary } i > 0, \text{ by (2).} \quad (5)$$

By (3), (5) also holds for $i = 0$; thus it holds for all $i \geq 0$. Therefore E is expansive and has the required approximating functions. \square

COROLLARY. If E is linearly expansive with respect to p_i and q_i , $i = 0, 1, \dots$, and f is the least function satisfying

$$f = E(f) \quad (\text{LE4})$$

then

$$f = p_0 \rightarrow q_0; \dots; p_n \rightarrow q_n; \dots \quad (\text{LE5})$$

12.9 The Algebra of Programs for the Lambda Calculus and for Combinators

Because Church's lambda calculus [5] and the system of combinators developed by Schönfinkel and Curry [6]

are the primary mathematical systems for representing the notion of application of functions, and because they are more powerful than FP systems, it is natural to enquire what an algebra of programs based on those systems would look like.

The lambda calculus and combinator equivalents of FP composition, $f \circ g$, are

$$\lambda fgx.(f(gx)) = B$$

where B is a simple combinator defined by Curry. There is no direct equivalent for the FP object $\langle x, y \rangle$ in the Church or Curry systems proper; however, following Landin [14] and Burge [4], one can use the primitive functions prefix, head, tail, null, and atomic to introduce the notion of list structures that correspond to FP sequences. Then, using FP notation for lists, the lambda calculus equivalent for construction is $\lambda fgx.\langle fx, gx \rangle$. A combinatory equivalent is an expression involving prefix, the null list, and two or more basic combinators. It is so complex that I shall not attempt to give it.

If one uses the lambda calculus or combinatory expressions for the functional forms $f \circ g$ and $[f, g]$ to express the law I.1 in the FP algebra, $[f, g] \circ h = [f \circ h, g \circ h]$, the result is an expression so complex that the sense of the law is obscured. The only way to make that sense clear in either system is to name the two functionals: composition = B , and construction = A , so that $Bfg = f \circ g$, and $Afg = [f, g]$. Then I.1 becomes

$$B(Afg)h = A(Bfh)(Bgh),$$

which is still not as perspicuous as the FP law.

The point of the above is that if one wishes to state clear laws like those of the FP algebra in either Church's or Curry's system, one finds it necessary to select certain functionals (e.g., composition and construction) as the basic operations of the algebra and to either give them short names or, preferably, represent them by some special notation as in FP. If one does this and provides primitives, objects, lists, etc., the result is an FP-like system in which the usual lambda expressions or combinators do not appear. Even then these Church or Curry versions of FP systems, being less restricted, have some problems that FP systems do not have:

a) The Church and Curry versions accommodate functions of many types and can define functions that do not exist in FP systems. Thus, Bf is a function that has no counterpart in FP systems. This added power carries with it problems of type compatibility. For example, in $f \circ g$, is the range of g included in the domain of f ? In FP systems all functions have the same domain and range.

b) The semantics of Church's lambda calculus depends on substitution rules that are simply stated but whose implications are very difficult to fully comprehend. The true complexity of these rules is not widely recognized but is evidenced by the succession of able logicians who have published "proofs" of the Church-Rosser theorem that failed to account for one or another

of these complexities. (The Church-Rosser theorem, or Scott's proof of the existence of a model [22], is required to show that the lambda calculus has a consistent semantics.) The definition of pure Lisp contained a related error for a considerable period (the "funarg" problem). Analogous problems attach to Curry's system as well.

In contrast, the formal (FFP) version of FP systems (described in the next section) has no variables and only an elementary substitution rule (a function for its name), and it can be shown to have a consistent semantics by a relatively simple fixed-point argument along the lines developed by Dana Scott and by Manna et al [16]. For such a proof see McJones [18].

12.10 Remarks

The algebra of programs outlined above needs much work to provide expansions for larger classes of equations and to extend its laws and theorems beyond the elementary ones given here. It would be interesting to explore the algebra for an FP-like system whose sequence constructor is not \perp -preserving (law I.5 is strengthened, but IV.1 is lost). Other interesting problems are: (a) Find rules that make expansions unique, giving canonical forms for functions; (b) find algorithms for expanding and analyzing the behavior of functions for various classes of arguments; and (c) explore ways of using the laws and theorems of the algebra as the basic rules either of a formal, preexecution "lazy evaluation" scheme [9, 10], or of one which operates during execution. Such schemes would, for example, make use of the law $I \circ [f, g] \leq f$ to avoid evaluating $g : x$.

13. Formal Systems for Functional Programming (FFP Systems)

13.1 Introduction

As we have seen, an FP system has a set of functions that depends on its set of primitive functions, its set of functional forms, and its set of definitions. In particular, its set of functional forms is fixed once and for all, and this set determines the power of the system in a major way. For example, if its set of functional forms is empty, then its entire set of functions is just the set of primitive functions. In FFP systems one can create new functional forms. Functional forms are represented by object sequences; the first element of a sequence determines which form it represents, while the remaining elements are the parameters of the form.

The ability to define new functional forms in FFP systems is one consequence of the principal difference between them and FP systems: in FFP systems objects are used to "represent" functions in a systematic way. Otherwise FFP systems mirror FP systems closely. They are similar to, but simpler than, the Reduction (Red) languages of an earlier paper [2].

We shall first give the simple syntax of FFP systems, then discuss their semantics informally, giving examples, and finally give their formal semantics.

13.2 Syntax

We describe the set O of objects and the set E of expressions of an FFP system. These depend on the choice of some set A of *atoms*, which we take as given. We assume that T (true), F (false), ϕ (the empty sequence), and $\#$ (default) belong to A , as well as "numbers" of various kinds, etc.

- 1) Bottom, \perp , is an *object* but not an atom.
- 2) Every atom is an *object*.
- 3) Every object is an *expression*.
- 4) If x_1, \dots, x_n are objects [expressions], then $\langle x_1, \dots, x_n \rangle$ is an *object* [resp., *expression*] called a *sequence* (of length n) for $n \geq 1$. The object [expression] x_i for $1 \leq i \leq n$, is the i th *element* of the sequence $\langle x_1, \dots, x_i, \dots, x_n \rangle$. (ϕ is both a sequence and an atom; its length is 0.)
- 5) If x and y are expressions, then $(x;y)$ is an *expression* called an *application*. x is its *operator* and y is its *operand*. Both are *elements* of the expression.
- 6) If $x = \langle x_1, \dots, x_n \rangle$ and if one of the elements of x is \perp , then $x = \perp$. That is, $\langle \dots, \perp, \dots \rangle = \perp$.
- 7) All objects and expressions are formed by finite use of the above rules.

A *subexpression* of an expression x is either x itself or a subexpression of an element of x . An FFP object is an expression that has no application as a subexpression. Given the same set of atoms, FFP and FP objects are the same.

13.3 Informal Remarks About FFP Semantics

13.3.1 The meaning of expressions; the semantic function μ . Every FFP expression e has a *meaning*, μe , which is always an object; μe is found by repeatedly replacing each innermost application in e by its meaning. If this process is nonterminating, the meaning of e is \perp . The meaning of an innermost application $(x;y)$ (since it is innermost, x and y must be objects) is the result of applying the function *represented* by x to y , just as in FP systems, except that in FFP systems functions are represented by objects, rather than by function expressions, with atoms (instead of function symbols) representing primitive and defined functions, and with sequences representing the FP functions denoted by functional forms.

The association between objects and the functions they represent is given by the *representation function*, ρ , of the FFP system. (Both ρ and μ belong to the description of the system, not the system itself.) Thus if the atom $NULL$ represents the FP function $null$, then $\rho NULL = null$ and the meaning of $(NULL:A)$ is $\mu(NULL:A) = (\rho NULL):A = null:A = F$.

From here on, as above, we use the colon in two senses. When it is between two objects, as in $(NULL:A)$, it identifies an FFP application that denotes only itself; when it comes between a *function* and an object, as in $(\rho NULL):A$ or $null:A$, it identifies an FP-like application that denotes the *result* of applying the function to the object.

The fact that FFP operators are objects makes pos-

sible a function, *apply*, which is meaningless in FP systems:

$$\text{apply}: \langle x, y \rangle = (x;y).$$

The result of $\text{apply}: \langle x, y \rangle$, namely $(x;y)$, is meaningless in FP systems on two levels. First, $(x;y)$ is not itself an object; it illustrates another difference between FP and FFP systems: some FFP functions, like *apply*, map objects into expressions, not directly into objects as FP functions do. However, the *meaning* of $\text{apply}: \langle x, y \rangle$ is an object (see below). Second, $(x;y)$ could not be even an intermediate result in an FP system; it is meaningless in FP systems since x is an object, not a function and FP systems do not associate functions with objects. Now if $APPLY$ represents *apply*, then the meaning of $(APPLY: \langle NULL, A \rangle)$ is

$$\begin{aligned} \mu(APPLY: \langle NULL, A \rangle) &= \mu((\rho APPLY): \langle NULL, A \rangle) \\ &= \mu(\text{apply}: \langle NULL, A \rangle) \\ &= \mu(NULL:A) = \mu((\rho NULL):A) \\ &= \mu(null:A) = \mu F = F. \end{aligned}$$

The last step follows from the fact that every object is its own meaning. Since the meaning function μ eventually evaluates all applications, one can think of $\text{apply}: \langle NULL, A \rangle$ as yielding F even though the actual result is $(NULL:A)$.

13.3.2 How objects represent functions; the representation function ρ . As we have seen, some atoms (*primitive atoms*) will represent the primitive functions of the system. Other atoms can represent defined functions just as symbols can in FP systems. If an atom is neither primitive nor defined, it represents \perp , the function which is \perp everywhere.

Sequences also represent functions and are analogous to the functional forms of FP. The function represented by a sequence is given (recursively) by the following rule.

Metacomposition rule

$$(\rho \langle x_1, \dots, x_n \rangle):y = (\rho x_1): \langle \langle x_1, \dots, x_n \rangle, y \rangle,$$

where the x_i 's and y are objects. Here ρx_1 determines what functional form $\langle x_1, \dots, x_n \rangle$ represents, and x_2, \dots, x_n are the parameters of the form (in FFP, x_1 itself can also serve as a parameter). Thus, for example, let $\text{Def } \rho CONST = 2 \circ 1$; then $\langle CONST, x \rangle$ in FFP represents the FP functional form λx , since, by the metacomposition rule, if $y \neq \perp$,

$$\begin{aligned} (\rho \langle CONST, x \rangle):y &= (\rho CONST): \langle \langle CONST, x \rangle, y \rangle \\ &= 2 \circ 1: \langle \langle CONST, x \rangle, y \rangle = x. \end{aligned}$$

Here we can see that the first, controlling, operator of a sequence or form, $CONST$ in this case, always has as its operand, after metacomposition, a pair whose first element is the sequence itself and whose second element is the original operand of the sequence, y in this case. The controlling operator can then rearrange and reapply the elements of the sequence and original operand in a great variety of ways. The significant point about metacom-

position is that it permits the definition of new functional forms, in effect, merely by defining new functions. It also permits one to write recursive functions without a definition.

We give one more example of a controlling function for a functional form: Def $\rho\text{CONS} = \alpha\text{apply}\circ\text{tl}\circ\text{distr}$. This definition results in $\langle\text{CONS}, f_1, \dots, f_n\rangle$ —where the f_i are objects—representing the same function as $[\rho f_1, \dots, \rho f_n]$. The following shows this.

$$\begin{aligned}
 & (\rho\langle\text{CONS}, f_1, \dots, f_n\rangle):x \\
 & \quad = (\rho\text{CONS}):<<\text{CONS}, f_1, \dots, f_n>, x> \\
 & \quad \quad \quad \text{by metacomposition} \\
 & = \alpha\text{apply}\circ\text{tl}\circ\text{distr}:<<\text{CONS}, f_1, \dots, f_n>, x> \\
 & \quad \quad \quad \text{by def of } \rho\text{CONS} \\
 & = \alpha\text{apply}:<<f_1, x>, \dots, <f_n, x>> \\
 & \quad \quad \quad \text{by def of tl and distr and } \circ \\
 & = <\text{apply}:<f_1, x>, \dots, \text{apply}:<f_n, x>> \\
 & \quad \quad \quad \text{by def of } \alpha \\
 & = <(f_1:x), \dots, (f_n:x)> \quad \text{by def of apply.}
 \end{aligned}$$

In evaluating the last expression, the meaning function μ will produce the meaning of each application, giving $\rho f_i:x$ as the i th element.

Usually, in describing the function represented by a sequence, we shall give its overall effect rather than show how its controlling operator achieves that effect. Thus we would simply write

$$(\rho\langle\text{CONS}, f_1, \dots, f_n\rangle):x = <(f_1:x), \dots, (f_n:x)>$$

instead of the more detailed account above.

We need a controlling operator, COMP , to give us sequences representing the functional form composition. We take ρCOMP to be a primitive function such that, for all objects x ,

$$\begin{aligned}
 & (\rho\langle\text{COMP}, f_1, \dots, f_n\rangle):x \\
 & \quad = (f_1:(f_2:(\dots:(f_n:x)\dots))) \quad \text{for } n \geq 1.
 \end{aligned}$$

(I am indebted to Paul McJones for his observation that ordinary composition could be achieved by this primitive function rather than by using two composition rules in the basic semantics, as was done in an earlier paper [2].)

Although FFP systems permit the definition and investigation of new functional forms, it is to be expected that most programming would use a fixed set of forms (whose controlling operators are primitives), as in FP, so that the algebraic laws for those forms could be employed, and so that a structured programming style could be used based on those forms.

In addition to its use in defining functional forms, metacomposition can be used to create recursive functions directly without the use of recursive definitions of the form Def $f = E(f)$. For example, if $\rho\text{MLAST} = \text{null}\circ\text{tl}\circ 2 \rightarrow 1\circ 2$; $\text{apply}\circ[1, \text{tl}\circ 2]$, then $\rho\langle\text{MLAST}\rangle = \text{last}$, where $\text{last}:x = x = \langle x_1, \dots, x_n \rangle \rightarrow x_n; \perp$. Thus the operator $\langle\text{MLAST}\rangle$ works as follows:

$$\mu\langle\text{MLAST}\rangle:<A, B>$$

$$\begin{aligned}
 & = \mu(\rho\text{MLAST}:<<\text{MLAST}>, <A, B>>) \\
 & \quad \quad \quad \text{by metacomposition} \\
 & = \mu(\text{apply}\circ[1, \text{tl}\circ 2]:<<\text{MLAST}>, <A, B>>) \\
 & = \mu(\text{apply}:<<\text{MLAST}>, >) \\
 & = \mu(\langle\text{MLAST}\rangle:) \\
 & = \mu(\rho\text{MLAST}:<<\text{MLAST}>, >) \\
 & = \mu(1\circ 2:<<\text{MLAST}>, >) \\
 & = B.
 \end{aligned}$$

13.3.3 Summary of the properties of ρ and μ . So far we have shown how ρ maps atoms and sequences into functions and how those functions map objects into expressions. Actually, ρ and all FFP functions can be extended so that they are defined for all expressions. With such extensions the properties of ρ and μ can be summarized as follows:

- 1) $\mu \in [\text{expressions} \rightarrow \text{objects}]$.
- 2) If x is an object, $\mu x = x$.
- 3) If e is an expression and $e = \langle e_1, \dots, e_n \rangle$, then $\mu e = \langle \mu e_1, \dots, \mu e_n \rangle$.
- 4) $\rho \in [\text{expressions} \rightarrow [\text{expressions} \rightarrow \text{expressions}]]$.
- 5) For any expression e , $\rho e = \rho(\mu e)$.
- 6) If x is an object and e an expression, then $\rho x:e = \rho x:(\mu e)$.
- 7) If x and y are objects, then $\mu(x;y) = \mu(\rho x;y)$. In words: the meaning of an FFP application $(x;y)$ is found by applying ρx , the function represented by x , to y and then finding the meaning of the resulting expression (which is usually an object and is then its own meaning).

13.3.4 Cells, fetching, and storing. For a number of reasons it is convenient to create functions which serve as names. In particular, we shall need this facility in describing the semantics of definitions in FFP systems. To introduce naming functions, that is, the ability to *fetch* the contents of a cell with a given name from a store (a sequence of cells) and to *store* a cell with given name and contents in such a sequence, we introduce objects called *cells* and two new functional forms, *fetch* and *store*.

Cells

A *cell* is a triple $\langle\text{CELL}, \text{name}, \text{contents}\rangle$. We use this form instead of the pair $\langle\text{name}, \text{contents}\rangle$ so that cells can be distinguished from ordinary pairs.

Fetch

The functional form *fetch* takes an object n as its parameter (n is customarily an atom serving as a name); it is written $\uparrow n$ (read “fetch n ”). Its definition for objects n and x is

$$\begin{aligned}
 \uparrow n:x &= x = \phi \rightarrow \#; \text{atom}:x \rightarrow \perp; \\
 (1:x) &= \langle\text{CELL}, n, c\rangle \rightarrow c; \uparrow n \circ \text{tl}:x,
 \end{aligned}$$

where $\#$ is the atom “default.” Thus $\uparrow n$ (fetch n) applied to a sequence gives the contents of the first cell in the sequence whose name is n ; If there is no cell named n , the result is default, $\#$. Thus $\uparrow n$ is the name function for the name n . (We assume that ρFETCH is the primitive function such that $\rho\langle\text{FETCH}, n\rangle = \uparrow n$. Note that $\uparrow n$ simply passes over elements in its operand that are not cells.)

Store and push, pop, purge

Like fetch, *store* takes an object n as its parameter; it is written $\downarrow n$ ("store n "). When applied to a pair $\langle x, y \rangle$, where y is a sequence, $\downarrow n$ removes the first cell named n from y , if any, then creates a new cell named n with contents x and appends it to y . Before defining $\downarrow n$ (store n) we shall specify four auxiliary functional forms. (These can be used in combination with fetch n and store n to obtain multiple, named, LIFO stacks within a storage sequence.) Two of these auxiliary forms are specified by recursive functional equations; each takes an object n as its parameter.

$$\begin{aligned} (\text{cellname } n) &\equiv \text{atom} \rightarrow \bar{F}; \\ &\quad \text{eq}^\circ[\text{length}, \bar{3}] \rightarrow \text{eq}^\circ[[\overline{\text{CELL}}, \bar{n}], [1, 2]]; \bar{F} \\ (\text{push } n) &\equiv \text{pair} \rightarrow \text{apndl}^\circ[[\overline{\text{CELL}}, \bar{n}, 1], 2]; \perp \\ (\text{pop } n) &\equiv \text{null} \rightarrow \bar{\phi}; \\ &\quad (\text{cellname } n)^\circ 1 \rightarrow \text{tl}; \text{apndl}^\circ[1, (\text{pop } n)^\circ \text{tl}] \\ (\text{purge } n) &\equiv \text{null} \rightarrow \bar{\phi}; (\text{cellname } n)^\circ 1 \rightarrow (\text{purge } n)^\circ \text{tl}; \\ &\quad \text{apndl}^\circ[1, (\text{purge } n)^\circ \text{tl}] \\ \downarrow n &\equiv \text{pair} \rightarrow (\text{push } n)^\circ[1, (\text{pop } n)^\circ 2]; \perp \end{aligned}$$

The above functional forms work as follows. For $x \neq \perp$, $(\text{cellname } n):x$ is T if x is a cell named n , otherwise it is F . $(\text{pop } n):y$ removes the first cell named n from a sequence y ; $(\text{purge } n):y$ removes all cells named n from y . $(\text{push } n):\langle x, y \rangle$ puts a cell named n with contents x at the head of sequence y ; $\downarrow n:\langle x, y \rangle$ is $(\text{push } n):\langle x, (\text{pop } n):y \rangle$.

(Thus $(\text{push } n):\langle x, y \rangle = y'$ pushes x onto the top of a "stack" named n in y' ; x can be read by $\uparrow n:y' = x$ and can be removed by $(\text{pop } n):y'$; thus $\uparrow n^\circ(\text{pop } n):y'$ is the element below x in the stack n , provided there is more than one cell named n in y' .)

13.3.5 Definitions in FFP systems. The semantics of an FFP system depends on a fixed set of definitions D (a sequence of cells), just as an FP system depends on its informally given set of definitions. Thus the semantic function μ depends on D ; altering D gives a new μ' that reflects the altered definitions. We have represented D as an *object* because in AST systems (Section 14) we shall want to transform D by applying functions to it and to fetch data from it—in addition to using it as the source of function definitions in FFP semantics.

If $\langle \text{CELL}, n, c \rangle$ is the first cell named n in the sequence D (and n is an atom) then it has the same effect as the FP definition $\text{Def } n \equiv pc$, that is, the meaning of $(n:x)$ will be the same as that of $pc:x$. Thus for example, if $\langle \text{CELL}, \text{CONST}, \langle \text{COMP}, 2, 1 \rangle \rangle$ is the first cell in D named CONST , then it has the same effect as $\text{Def } \text{CONST} = 2 \circ 1$, and the FFP system with that D would find

$$\mu(\text{CONST}: \langle \langle x, y \rangle, z \rangle) = y$$

and consequently

$$\mu(\langle \text{CONST}, A \rangle: B) = A.$$

In general, in an FFP system with definitions D , the meaning of an application of the form $(\text{atom}:x)$ is de-

pendent on D ; if $\uparrow \text{atom}:D \neq \#$ (that is, atom is defined in D) then its meaning is $\mu(c:x)$, where $c = \uparrow \text{atom}:D$, the contents of the first cell in D named atom . If $\uparrow \text{atom}:D = \#$, then atom is not defined in D and either atom is primitive, i.e. the system knows how to compute $\text{patom}:x$, and $\mu(\text{atom}:x) = \mu(\text{patom}:x)$, otherwise $\mu(\text{atom}:x) = \perp$.

13.4 Formal Semantics for FFP Systems

We assume that a set A of atoms, a set D of definitions, a set $P \subset A$ of primitive atoms and the primitive functions they represent have all been chosen. We assume that ρa is the primitive function represented by a if a belongs to P , and that $\rho a = \perp$ if a belongs to Q , the set of atoms in $A - P$ that are not defined in D . Although ρ is defined for all expressions (see 13.3.3), the formal semantics uses its definition only on P and Q . The functions that ρ assigns to other expressions x are implicitly determined and applied in the following semantic rules for evaluating $\mu(x:y)$. The above choices of A and D , and of P and the associated primitive functions determine the objects, expressions, and the semantic function μ_D for an FFP system. (We regard D as fixed and write μ for μ_D .) We assume D is a sequence and that $\uparrow y:D$ can be computed (by the function $\uparrow y$ as given in Section 13.3.4) for any atom y . With these assumptions we define μ as the least fixed point of the functional τ , where the function $\tau\mu$ is defined as follows for any function μ (for all expressions x, x_i, y, y_i, z , and w):

$$\begin{aligned} (\tau\mu)x &\equiv x \in A \rightarrow x; \\ x &= \langle x_1, \dots, x_n \rangle \rightarrow \langle \mu x_1, \dots, \mu x_n \rangle; \\ x &= \langle y:z \rangle \rightarrow \\ &\quad (y \in A \& (\uparrow y:D) = \# \rightarrow \mu((\rho y)(\mu z))); \\ &\quad y \in A \& (\uparrow y:D) = w \rightarrow \mu(w:z); \\ &\quad y = \langle y_1, \dots, y_n \rangle \rightarrow \mu(y_1: \langle y, z \rangle); \mu(\mu y:z); \perp \end{aligned}$$

The above description of μ expands the operator of an application by definitions and by metacomposition before evaluating the operand. It is assumed that predicates like " $x \in A$ " in the above definition of $\tau\mu$ are \perp -preserving (e.g., " $\perp \in A$ " has the value \perp) and that the conditional expression itself is also \perp -preserving. Thus $(\tau\mu)\perp = \perp$ and $(\tau\mu)(\perp:z) = \perp$. This concludes the semantics of FFP systems.

14. Applicative State Transition Systems (AST Systems)

14.1 Introduction

This section sketches a class of systems mentioned earlier as alternatives to von Neumann systems. It must be emphasized again that these applicative state transition systems are put forward not as practical programming systems in their present form, but as examples of a class in which applicative style programming is made available in a history sensitive, but non-von Neumann system. These systems are loosely coupled to states and depend on an underlying applicative system for both

their programming language and the description of their state transitions. The underlying applicative system of the AST system described below is an FFP system, but other applicative systems could also be used.

To understand the reasons for the structure of AST systems, it is helpful first to review the basic structure of a von Neumann system, Algol, observe its limitations, and compare it with the structure of AST systems. After that review a minimal AST system is described; a small, top-down, self-protecting system program for file maintenance and running user programs is given, with directions for installing it in the AST system and for running an example user program. The system program uses "name functions" instead of conventional names and the user may do so too. The section concludes with subsections discussing variants of AST systems, their general properties, and naming systems.

14.2 The Structure of Algol Compared to That of AST Systems

An Algol program is a sequence of statements, each representing a transformation of the Algol state, which is a complex repository of information about the status of various stacks, pointers, and variable mappings of identifiers onto values, etc. Each statement communicates with this constantly changing state by means of complicated protocols peculiar to itself and even to its different parts (e.g., the protocol associated with the variable x depends on its occurrence on the left or right of an assignment, in a declaration, as a parameter, etc.).

It is as if the Algol state were a complex "store" that communicates with the Algol program through an enormous "cable" of many specialized wires. The complex communications protocols of this cable are fixed and include those for every statement type. The "meaning" of an Algol program must be given in terms of the total effect of a vast number of communications with the state via the cable and its protocols (plus a means for identifying the output and inserting the input into the state). By comparison with this massive cable to the Algol state/store, the cable that is the von Neumann bottleneck of a computer is a simple, elegant concept.

Thus Algol statements are not expressions representing state-to-state functions that are built up by the use of orderly combining forms from simpler state-to-state functions. Instead they are complex *messages* with context-dependent parts that nibble away at the state. Each part transmits information to and from the state over the cable by its own protocols. There is no provision for applying general functions to the *whole* state and thereby making large changes in it. The possibility of large, powerful transformations of the state S by function application, $S \rightarrow f.S$, is in fact inconceivable in the von Neumann—cable and protocol—context: there could be no assurance that the new state $f.S$ would match the cable and its fixed protocols unless f is restricted to the tiny changes allowed by the cable in the first place.

We want a computing system whose semantics does

not depend on a host of baroque protocols for communicating with the state, and we want to be able to make large transformations in the state by the application of general functions. AST systems provide one way of achieving these goals. Their semantics has two protocols for getting information from the state: (1) get from it the definition of a function to be applied, and (2) get the whole state itself. There is one protocol for changing the state: compute the new state by function application. Besides these communications with the state, AST semantics is applicative (i.e. FFP). It does not depend on state changes because the state does not change at all during a computation. Instead, the result of a computation is output *and* a new state. The structure of an AST state is slightly restricted by one of its protocols: It must be possible to identify a definition (i.e. cell) in it. Its structure—it is a sequence—is far simpler than that of the Algol state.

Thus the structure of AST systems avoids the complexity and restrictions of the von Neumann state (with its communications protocols) while achieving greater power and freedom in a radically different and simpler framework.

14.3 Structure of an AST System

An AST system is made up of three elements:

- 1) An *applicative subsystem* (such as an FFP system).
- 2) A *state D* that is the set of definitions of the applicative subsystem.
- 3) A set of *transition rules* that describe how inputs are transformed into outputs and how the state D is changed.

The programming language of an AST system is just that of its applicative subsystem. (From here on we shall assume that the latter is an FFP system.) Thus AST systems can use the FP programming style we have discussed. The applicative subsystem cannot change the state D and it does not change during the evaluation of an expression. A new state is computed along with output and replaces the old state when output is issued. (Recall that a set of definitions D is a sequence of cells; a cell name is the name of a defined function and its contents is the defining expression. Here, however, some cells may name data rather than functions; a data name n will be used in $\uparrow n$ (fetch n) whereas a function name will be used as an operator itself.)

We give below the transition rules for the elementary AST system we shall use for examples of programs. These are perhaps the simplest of many possible transition rules that could determine the behavior of a great variety of AST systems.

14.3.1 Transition rules for an elementary AST system. When the system receives an input x , it forms the application (*SYSTEM*: x) and then proceeds to obtain its meaning in the FFP subsystem, using the current state D as the set of definitions. *SYSTEM* is the distinguished name of a function defined in D (i.e. it is the "system program"). Normally the result is a pair

$$\mu(SYSTEM:x) = \langle o, d \rangle$$

where o is the system output that results from input x and d becomes the new state D for the system's next input. Usually d will be a copy or partly changed copy of the old state. If $\mu(SYSTEM:x)$ is not a pair, the output is an error message and the state remains unchanged.

14.3.2 Transition rules: exception conditions and startup. Once an input has been accepted, our system will not accept another (except $\langle RESET, x \rangle$, see below) until an output has been issued and the new state, if any, installed. The system will accept the input $\langle RESET, x \rangle$ at any time. There are two cases: (a) If $SYSTEM$ is defined in the current state D , then the system aborts its current computation without altering D and treats x as a new normal input; (b) if $SYSTEM$ is not defined in D , then x is appended to D as its first element. (This ends the complete description of the transition rules for our elementary AST system.)

If $SYSTEM$ is defined in D it can always prevent any change in its own definition. If it is not defined, an ordinary input x will produce $\mu(SYSTEM:x) = \perp$ and the transition rules yield an error message and an unchanged state; on the other hand, the input $\langle RESET, \langle CELL, SYSTEM, s \rangle \rangle$ will define $SYSTEM$ to be s .

14.3.3 Program access to the state; the function $\rho DEFS$. Our FFP subsystem is required to have one new primitive function, $defs$, named $DEFS$ such that for any object $x \neq \perp$,

$$defs:x = \rho DEFS:x = D$$

where D is the current state and set of definitions of the AST system. This function allows programs access to the whole state for any purpose, including the essential one of computing the successor state.

14.4 An Example of a System Program

The above description of our elementary AST system, plus the FFP subsystem and the FP primitives and functional forms of earlier sections, specify a complete history-sensitive computing system. Its input and output behavior is limited by its simple transition rules, but otherwise it is a powerful system once it is equipped with a suitable set of definitions. As an example of its use we shall describe a small system program, its installation, and operation.

Our example system program will handle queries and updates for a file it maintains, evaluate FFP expressions, run general user programs that do not damage the file or the state, and allow authorized users to change the set of definitions and the system program itself. All inputs it accepts will be of the form $\langle key, input \rangle$ where key is a code that determines both the input class (*system-change*, *expression*, *program*, *query*, *update*) and also the identity of the user and his authority to use the system for the given input class. We shall not specify a format for key . $Input$ is the input itself, of the class given by key .

14.4.1 General plan of the system program. The state

D of our AST system will contain the definitions of all nonprimitive functions needed for the system program and for users' programs. (Each definition is in a cell of the sequence D .) In addition, there will be a cell in D named *FILE* with contents *file*, which the system maintains. We shall give FP definitions of functions and later show how to get them into the system in their FFP form. The transition rules make the input the operand of *SYSTEM*, but our plan is to use name-functions to refer to data, so the first thing we shall do with the input is to create two cells named *KEY* and *INPUT* with contents *key* and *input* and append these to D . This sequence of cells has one each for *key*, *input*, and *file*; it will be the operand of our main function called *subsystem*. Subsystem can then obtain *key* by applying $\uparrow KEY$ to its operand, etc. Thus the definition

$$\text{Def } \text{system} = \text{pair} \rightarrow \text{subsystem} \circ f, [\text{NONPAIR}, \text{defs}]$$

where

$$f = \downarrow INPUT \circ [2, \downarrow KEY \circ [1, \text{defs}]]$$

causes the system to output *NONPAIR* and leave the state unchanged if the input is not a pair. Otherwise, if it is $\langle key, input \rangle$, then

$$f: \langle key, input \rangle = \langle \langle \langle \text{CELL}, \text{INPUT}, \text{input} \rangle, \langle \text{CELL}, \text{KEY}, \text{key} \rangle, d_1, \dots, d_n \rangle \rangle$$

where $D = \langle d_1, \dots, d_n \rangle$. (We might have constructed a different operand than the one above, one with just three cells, for *key*, *input*, and *file*. We did not do so because real programs, unlike *subsystem*, would contain many name functions referring to data in the state, and this "standard" construction of the operand would suffice then as well.)

14.4.2 The "subsystem" function. We now give the FP definition of the function *subsystem*, followed by brief explanations of its six cases and auxiliary functions.

$$\begin{aligned} \text{Def } \text{subsystem} = & \text{is-system-change} \circ \uparrow KEY \rightarrow [\text{report-change}, \text{apply}] \circ [\uparrow INPUT, \text{defs}]; \\ & \text{is-expression} \circ \uparrow KEY \rightarrow [\uparrow INPUT, \text{defs}]; \\ & \text{is-program} \circ \uparrow KEY \rightarrow [\text{system-check}, \text{apply}] \circ [\uparrow INPUT, \text{defs}]; \\ & \text{is-query} \circ \uparrow KEY \rightarrow [\text{query-response}] \circ [\uparrow INPUT, \uparrow FILE], \text{defs}; \\ & \text{is-update} \circ \uparrow KEY \rightarrow \\ & \quad [\text{report-update}, \downarrow FILE \circ [\text{update}, \text{defs}]] \\ & \quad \circ [\uparrow INPUT, \uparrow FILE]; \\ & \quad [\text{report-error}] \circ [\uparrow KEY, \uparrow INPUT], \text{defs}. \end{aligned}$$

This subsystem has five " $p \rightarrow f$ " clauses and a final default function, for a total of six classes of inputs; the treatment of each class is given below. Recall that the *operand* of *subsystem* is a sequence of cells containing *key*, *input*, and *file* as well as all the defined functions of D , and that *subsystem:operand* = $\langle output, newstate \rangle$.

Default inputs. In this case the result is given by the last (default) function of the definition when key does not satisfy any of the preceding clauses. The output is *report-error*: $\langle key, input \rangle$. The state is unchanged since it is given by *defs:operand* = D . (We leave to the reader's imagination what the function *report-error* will generate from its *operand*.)

System-change inputs. When

is-system-change $\uparrow KEY:operand =$

is-system-change $:key = T,$

key specifies that the user is authorized to make a system change and that $input = \uparrow INPUT:operand$ represents a function f that is to be applied to D to produce the new state $f:D$. (Of course $f:D$ can be a useless new state; no constraints are placed on it.) The output is a report, namely $report-change:\langle input,D \rangle$.

Expression inputs. When $is-expression:key = T$, the system understands that the output is to be the meaning of the FFP expression $input$; $\uparrow INPUT:operand$ produces it and it is evaluated, as are all expressions. The state is unchanged.

Program inputs and system self-protection. When $is-program:key = T$, both the output and new state are given by $(input):D = \langle output,newstate \rangle$. If *newstate* contains *file* in suitable condition and the definitions of system and other protected functions, then
 $system-check:\langle output,newstate \rangle = \langle output,newstate \rangle$. Otherwise, $system-check:\langle output,newstate \rangle = \langle error-report,D \rangle$.

Although *program* inputs can make major, possibly disastrous changes in the state when it produces *newstate*, *system-check* can use any criteria to either allow it to become the actual new state or to keep the old. A more sophisticated *system-check* might correct only prohibited changes in the state. Functions of this sort are possible because they can always access the old state for comparison with the new state-to-be and control what state transition will finally be allowed.

File query inputs. If $is-query:key = T$, the function *query-response* is designed to produce the output = answer to the query *input* from its operand $\langle input,file \rangle$.

File update inputs. If $is-update:key = T$, *input* specifies a file transaction understood by the function *update*, which computes $updated-file = update:\langle input,file \rangle$. Thus $\downarrow FILE$ has $\langle updated-file,D \rangle$ as its operand and thus stores the updated file in the cell *FILE* in the new state. The rest of the state is unchanged. The function *report-update* generates the output from its operand $\langle input,file \rangle$.

14.4.3 Installing the system program. We have described the function called *system* by some FP definitions (using auxiliary functions whose behavior is only indicated). Let us suppose that we have FP definitions for all the nonprimitive functions required. Then each definition can be converted to give the name and contents of a cell in D (of course this conversion itself would be done by a better system). The conversion is accomplished by changing each FP function name to its equivalent atom (e.g., *update* becomes *UPDATE*) and by replacing functional forms by sequences whose first member is the controlling function for the particular form. Thus $\downarrow FILE \circ [update,defs]$ is converted to

$\langle COMP, \langle STORE,FILE \rangle,$

$\langle CONS, UPDATE, DEFS \rangle \rangle,$

and the FP function is the same as that represented by the FFP object, provided that $update = \rho UPDATE$ and *COMP*, *STORE*, and *CONS* represent the controlling functions for composition, store, and construction.

All FP definitions needed for our system can be converted to cells as indicated above, giving a sequence D_0 . We assume that the AST system has an empty state to start with, hence *SYSTEM* is not defined. We want to define *SYSTEM* initially so that it will install its next input as the state; having done so we can then input D_0 and all our definitions will be installed, including our program—system—itself. To accomplish this we enter our first input

$\langle RESET, \langle CELL, SYSTEM, loader \rangle \rangle$

where $loader = \langle CONS, \langle CONST, DONE \rangle, ID \rangle$.

Then, by the transition rule for *RESET* when *SYSTEM* is undefined in D , the cell in our input is put at the head of $D = \phi$, thus defining $\rho SYSTEM = ploader = [DONE, id]$. Our second input is D_0 , the set of definitions we wish to become the state. The regular transition rule causes the AST system to evaluate $\mu(SYSTEM:D_0) = [DONE, id]:D_0 = \langle DONE, D_0 \rangle$. Thus the output from our second input is *DONE*, the new state is D_0 , and $\rho SYSTEM$ is now our system program (which only accepts inputs of the form $\langle key, input \rangle$).

Our next task is to load the file (we are given an initial value *file*). To load it we input a *program* into the newly installed system that contains *file* as a constant and stores it in the state; the input is

$\langle program-key, [DONE, store-file] \rangle$ where

$pstore-file = \downarrow FILE \circ [file, id]$.

Program-key identifies $[DONE, store-file]$ as a program to be applied to the state D_0 to give the output and new state D_1 , which is:

$pstore-file:D_0 = \downarrow FILE \circ [file, id]:D_0$,

or D_0 with a cell containing *file* at its head. The output is $\overline{DONE}:D_0 = DONE$. We assume that *system-check* will pass $\langle DONE, D_1 \rangle$ unchanged. FP expressions have been used in the above in place of the FFP objects they denote, e.g. *DONE* for $\langle CONST, DONE \rangle$.

14.4.4 Using the system. We have not said how the system's file, queries or updates are structured, so we cannot give a detailed example of file operations. However, the structure of subsystem shows clearly how the system's response to queries and updates depends on the functions *query-response*, *update*, and *report-update*.

Let us suppose that matrices m , n named *M*, and *N* are stored in D and that the function *MM* described earlier is defined in D . Then the input

$\langle expression-key, (MM \circ [\uparrow M, \uparrow N] \circ DEFS:\#) \rangle$

would give the product of the two matrices as output and an unchanged state. *Expression-key* identifies the application as an expression to be evaluated and since $defs:\# = D$ and $[\uparrow M, \uparrow N]:D = \langle m, n \rangle$, the value of the expression is the result $MM:\langle m, n \rangle$, which is the output.

Our miniature system program has no provision for giving control to a user's program to process many inputs, but it would not be difficult to give it that capability while still monitoring the user's program with the option of taking control back.

14.5 Variants of AST Systems

A major extension of the AST systems suggested above would provide combining forms, "system forms," for building a new AST system from simpler, component AST systems. That is, a system form would take AST systems as parameters and generate a new AST system, just as a functional form takes functions as parameters and generates new functions. These system forms would have properties like those of functional forms and would become the "operations" of a useful "algebra of systems" in much the same way that functional forms are the "operations" of the algebra of programs. However, the problem of finding useful system forms is much more difficult, since they must handle *RESETS*, match inputs and outputs, and combine history-sensitive systems rather than fixed functions.

Moreover, the usefulness or need for system forms is less clear than that for functional forms. The latter are essential for building a great variety of functions from an initial primitive set, whereas, even without system forms, the facilities for building AST systems are already so rich that one could build virtually any system (with the general input and output properties allowed by the given AST scheme). Perhaps system forms would be useful for building systems with complex input and output arrangements.

14.6 Remarks About AST Systems

As I have tried to indicate above, there can be innumerable variations in the ingredients of an AST system—how it operates, how it deals with input and output, how and when it produces new states, and so on. In any case, a number of remarks apply to any reasonable AST system:

a) A state transition occurs once per major computation and can have useful mathematical properties. State transitions are not involved in the tiniest details of a computation as in conventional languages; thus the linguistic von Neumann bottleneck has been eliminated. No complex "cable" or protocols are needed to communicate with the state.

b) Programs are written in an applicative language that can accommodate a great range of changeable parts, parts whose power and flexibility exceed that of any von Neumann language so far. The word-at-a-time style is replaced by an applicative style; there is no division of programming into a world of expressions and a world of statements. Programs can be analyzed and optimized by an algebra of programs.

c) Since the state cannot change during the computation of *system:x*, there are no side effects. Thus independent applications can be evaluated in parallel.

d) By defining appropriate functions one can, I believe, introduce major new features at any time, using the same framework. Such features must be built into the framework of a von Neumann language. I have in mind such features as: "stores" with a great variety of naming systems, types and type checking, communicating parallel processes, nondeterminacy and Dijkstra's "guarded command" constructs [8], and improved methods for structured programming.

e) The framework of an AST system comprises the syntax and semantics of the underlying applicative system plus the system framework sketched above. By current standards, this is a tiny framework for a language and is the only fixed part of the system.

14.7 Naming Systems in AST and von Neumann Models

In an AST system, naming is accomplished by functions as indicated in Section 13.3.3. Many useful functions for altering and accessing a store can be defined (e.g. push, pop, purge, typed fetch, etc.). All these definitions and their associated naming systems can be introduced without altering the AST framework. Different kinds of "stores" (e.g., with "typed cells") with individual naming systems can be used in one program. A cell in one store may contain another entire store.

The important point about AST naming systems is that they utilize the functional nature of names (Reynolds' GEDÄNKEN [19] also does so to some extent within a von Neumann framework). Thus name functions can be composed and combined with other functions by functional forms. In contrast, functions and names in von Neumann languages are usually disjoint concepts and the function-like nature of names is almost totally concealed and useless, because a) names cannot be applied as functions; b) there are no general means to combine names with other names and functions; c) the objects to which name functions apply (stores) are not accessible as objects.

The failure of von Neumann languages to treat names as functions may be one of their more important weaknesses. In any case, the ability to use names as functions and stores as objects may turn out to be a useful and important programming concept, one which should be thoroughly explored.

15. Remarks About Computer Design

The dominance of von Neumann languages has left designers with few intellectual models for practical computer designs beyond variations of the von Neumann computer. Data flow models [1] [7] [13] are one alternative class of history-sensitive models. The substitution rules of lambda-calculus based languages present serious problems for the machine designer. Berkling [3] has developed a modified lambda calculus that has three kinds of applications and that makes renaming of vari-

ables unnecessary. He has developed a machine to evaluate expressions of this language. Further experience is needed to show how sound a basis this language is for an effective programming style and how efficient his machine can be.

Magó [15] has developed a novel applicative machine built from identical components (of two kinds). It evaluates, directly, FP-like and other applicative expressions from the bottom up. It has no von Neumann store and no address register, hence no bottleneck; it is capable of evaluating many applications in parallel; its built-in operations resemble FP operators more than von Neumann computer operations. It is the farthest departure from the von Neumann computer that I have seen.

There are numerous indications that the applicative style of programming can become more powerful than the von Neumann style. Therefore it is important for programmers to develop a new class of history-sensitive models of computing systems that embody such a style and avoid the inherent efficiency problems that seem to attach to lambda-calculus based systems. Only when these models and their applicative languages have proved their superiority over conventional languages will we have the economic basis to develop the new kind of computer that can best implement them. Only then, perhaps, will we be able to fully utilize large-scale integrated circuits in a computer design not limited by the von Neumann bottleneck.

16. Summary

The fifteen preceding sections of this paper can be summarized as follows.

Section 1. Conventional programming languages are large, complex, and inflexible. Their limited expressive power is inadequate to justify their size and cost.

Section 2. The models of computing systems that underlie programming languages fall roughly into three classes: (a) simple operational models (e.g., Turing machines), (b) applicative models (e.g., the lambda calculus), and (c) von Neumann models (e.g., conventional computers and programming languages). Each class of models has an important difficulty: The programs of class (a) are inscrutable; class (b) models cannot save information from one program to the next; class (c) models have unusable foundations and programs that are conceptually unhelpful.

Section 3. Von Neumann computers are built around a bottleneck: the word-at-a-time tube connecting the CPU and the store. Since a program must make its overall change in the store by pumping vast numbers of words back and forth through the von Neumann bottleneck, we have grown up with a style of programming that concerns itself with this word-at-a-time traffic through the bottleneck rather than with the larger conceptual units of our problems.

Section 4. Conventional languages are based on the

programming style of the von Neumann computer. Thus variables = storage cells; assignment statements = fetching, storing, and arithmetic; control statements = jump and test instructions. The symbol “=” is the linguistic von Neumann bottleneck. Programming in a conventional—von Neumann—language still concerns itself with the word-at-a-time traffic through this slightly more sophisticated bottleneck. Von Neumann languages also split programming into a world of expressions and a world of statements; the first of these is an orderly world, the second is a disorderly one, a world that structured programming has simplified somewhat, but without attacking the basic problems of the split itself and of the word-at-a-time style of conventional languages.

Section 5. This section compares a von Neumann program and a functional program for inner product. It illustrates a number of problems of the former and advantages of the latter: e.g., the von Neumann program is repetitive and word-at-a-time, works only for two vectors named *a* and *b* of a given length *n*, and can only be made general by use of a procedure declaration, which has complex semantics. The functional program is nonrepetitive, deals with vectors as units, is more hierarchically constructed, is completely general, and creates “housekeeping” operations by composing high-level housekeeping operators. It does not name its arguments, hence it requires no procedure declaration.

Section 6. A programming language comprises a framework plus some changeable parts. The framework of a von Neumann language requires that most features must be built into it; it can accommodate only limited changeable parts (e.g., user-defined procedures) because there must be detailed provisions in the “state” and its transition rules for all the needs of the changeable parts, as well as for all the features built into the framework. The reason the von Neumann framework is so inflexible is that its semantics is too closely coupled to the state: every detail of a computation changes the state.

Section 7. The changeable parts of von Neumann languages have little expressive power; this is why most of the language must be built into the framework. The lack of expressive power results from the inability of von Neumann languages to effectively use combining forms for building programs, which in turn results from the split between expressions and statements. Combining forms are at their best in expressions, but in von Neumann languages an expression can only produce a single word; hence expressive power in the world of expressions is mostly lost. A further obstacle to the use of combining forms is the elaborate use of naming conventions.

Section 8. APL is the first language not based on the lambda calculus that is not word-at-a-time and uses functional combining forms. But it still retains many of the problems of von Neumann languages.

Section 9. Von Neumann languages do not have useful properties for reasoning about programs. Axiomatic and denotational semantics are precise tools for describing and understanding conventional programs,

but they only talk about them and cannot alter their ungainly properties. Unlike von Neumann languages, the language of ordinary algebra is suitable both for stating its laws and for transforming an equation into its solution, all within the "language."

Section 10. In a history-sensitive language, a program can affect the behavior of a subsequent one by changing some store which is saved by the system. Any such language requires some kind of state transition semantics. But it does not need semantics closely coupled to states in which the state changes with every detail of the computation. "Applicative state transition" (AST) systems are proposed as history-sensitive alternatives to von Neumann systems. These have: (a) loosely coupled state-transition semantics in which a transition occurs once per major computation; (b) simple states and transition rules; (c) an underlying applicative system with simple "reduction" semantics; and (d) a programming language and state transition rules both based on the underlying applicative system and its semantics. The next four sections describe the elements of this approach to non-von Neumann language and system design.

Section 11. A class of informal functional programming (FP) systems is described which use no variables. Each system is built from objects, functions, functional forms, and definitions. Functions map objects into objects. Functional forms combine existing functions to form new ones. This section lists examples of primitive functions and functional forms and gives sample programs. It discusses the limitations and advantages of FP systems.

Section 12. An "algebra of programs" is described whose variables range over the functions of an FP system and whose "operations" are the functional forms of the system. A list of some twenty-four laws of the algebra is followed by an example proving the equivalence of a nonrepetitive matrix multiplication program and a recursive one. The next subsection states the results of two "expansion theorems" that "solve" two classes of equations. These solutions express the "unknown" function in such equations as an infinite conditional expansion that constitutes a case-by-case description of its behavior and immediately gives the necessary and sufficient conditions for termination. These results are used to derive a "recursion theorem" and an "iteration theorem," which provide ready-made expansions for some moderately general and useful classes of "linear" equations. Examples of the use of these theorems treat: (a) correctness proofs for recursive and iterative factorial functions, and (b) a proof of equivalence of two iterative programs. A final example deals with a "quadratic" equation and proves that its solution is an idempotent function. The next subsection gives the proofs of the two expansion theorems.

The algebra associated with FP systems is compared with the corresponding algebras for the lambda calculus and other applicative systems. The comparison shows some advantages to be drawn from the severely restricted

FP systems, as compared with the much more powerful classical systems. Questions are suggested about algorithmic reduction of functions to infinite expansions and about the use of the algebra in various "lazy evaluation" schemes.

Section 13. This section describes formal functional programming (FFP) systems that extend and make precise the behavior of FP systems. Their semantics are simpler than that of classical systems and can be shown to be consistent by a simple fixed-point argument.

Section 14. This section compares the structure of Algol with that of applicative state transition (AST) systems. It describes an AST system using an FFP system as its applicative subsystem. It describes the simple state and the transition rules for the system. A small self-protecting system program for the AST system is described, and how it can be installed and used for file maintenance and for running user programs. The section briefly discusses variants of AST systems and functional naming systems that can be defined and used within an AST system.

Section 15. This section briefly discusses work on applicative computer designs and the need to develop and test more practical models of applicative systems as the future basis for such designs.

Acknowledgments. In earlier work relating to this paper I have received much valuable help and many suggestions from Paul R. McJones and Barry K. Rosen. I have had a great deal of valuable help and feedback in preparing this paper. James N. Gray was exceedingly generous with his time and knowledge in reviewing the first draft. Stephen N. Zilles also gave it a careful reading. Both made many valuable suggestions and criticisms at this difficult stage. It is a pleasure to acknowledge my debt to them. I also had helpful discussions about the first draft with Ronald Fagin, Paul R. McJones, and James H. Morris, Jr. Fagin suggested a number of improvements in the proofs of theorems.

Since a large portion of the paper contains technical material, I asked two distinguished computer scientists to referee the third draft. David J. Gries and John C. Reynolds were kind enough to accept this burdensome task. Both gave me large, detailed sets of corrections and overall comments that resulted in many improvements, large and small, in this final version (which they have not had an opportunity to review). I am truly grateful for the generous time and care they devoted to reviewing this paper.

Finally, I also sent copies of the third draft to Gyula A. Magó, Peter Naur, and John H. Williams. They were kind enough to respond with a number of extremely helpful comments and corrections. Geoffrey A. Frank and Dave Tolle at the University of North Carolina reviewed Magó's copy and pointed out an important error in the definition of the semantic function of FFP systems. My grateful thanks go to all these kind people for their help.

References

1. Arvind, and Gostelow, K.P. A new interpreter for data flow schemas and its implications for computer architecture. Tech. Rep. No. 72, Dept. Comptr. Sci., U. of California, Irvine, Oct. 1975.
2. Backus, J. Programming language semantics and closed applicative languages. Conf. Record ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, 71-86.
3. Berkling, K.J. Reduction languages for reduction machines. Interner Bericht ISF-76-8, Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn, Sept. 1976.
4. Burge, W.H. *Recursive Programming Techniques*. Addison-Wesley, Reading, Mass., 1975.
5. Church, A. *The Calculi of Lambda-Conversion*. Princeton U. Press, Princeton, N.J., 1941.
6. Curry, H.B., and Feys, R. *Combinatory Logic, Vol. I*. North-Holland Pub. Co., Amsterdam, 1958.
7. Dennis, J.B. First version of a data flow procedure language. Tech. Mem. No. 61, Lab. for Comptr. Sci., M.I.T., Cambridge, Mass., May 1973.
8. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
9. Friedman, D.P., and Wise, D.S. CONS should not evaluate its arguments. In *Automata, Languages and Programming*, S. Michaelson and R. Milner, Eds., Edinburgh U. Press, Edinburgh, 1976, pp. 257-284.
10. Henderson, P., and Morris, J.H. Jr. A lazy evaluator. Conf. Record Third ACM Symp. on Principles of Programming Languages, Atlanta, Ga., Jan. 1976, pp. 95-103.
11. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM* 12, 10 (Oct. 1969), 576-583.
12. Iverson, K. *A Programming Language*. Wiley, New York, 1962.
13. Kosinski, P. A data flow programming language. Rep. RC 4264, IBM T.J. Watson Research Ctr., Yorktown Heights, N.Y., March 1973.
14. Landin, P.J. The mechanical evaluation of expressions. *Computer J.* 6, 4 (1964), 308-320.
15. Magó, G.A. A network of microprocessors to execute reduction languages. To appear in *Int. J. Comprtr. and Inform. Sci.*
16. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. *Comm. ACM* 16, 8 (Aug. 1973) 491-502.
17. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, Pt. 1. *Comm. ACM* 3, 4 (April 1960), 184-195.
18. McJones, P. A Church-Rosser property of closed applicative languages. Rep. RJ 1589, IBM Res. Lab., San Jose, Calif., May 1975.
19. Reynolds, J.C. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM* 13, 5 (May 1970), 308-318.
20. Reynolds, J.C. Notes on a lattice-theoretic approach to the theory of computation. Dept. Syst. and Inform. Sci., Syracuse U., Syracuse, N.Y., 1972.
21. Scott, D. Outline of a mathematical theory of computation. Proc. 4th Princeton Conf. on Inform. Sci. and Syst., 1970.
22. Scott, D. Lattice-theoretic models for various type-free calculi. Proc. Fourth Int. Congress for Logic, Methodology, and the Philosophy of Science, Bucharest, 1972.
23. Scott, D., and Strachey, C. Towards a mathematical semantics for computer languages. Proc. Symp. on Comptrs. and Automata, Polytechnic Inst. of Brooklyn, 1971.

MICROPIPELINES

IVAN E. SUTHERLAND

The pipeline processor is a common paradigm for very high speed computing machinery. Pipeline processors provide high speed because their separate stages can operate concurrently, much as different people on a manufacturing assembly line work concurrently on material passing down the line. Although the concurrency of pipeline processors makes their design a demanding task, they can be found in graphics processors, in signal processing devices, in integrated circuit components for doing arithmetic, and in the instruction interpretation units and arithmetic operations of general purpose computing machinery.

Because I plan to describe a variety of pipeline processors, I will start by suggesting names for their various forms. Pipeline processors, or more simply just pipelines, operate on data as it passes along them. The latency of a pipeline is a measure of how long it takes a single data value to pass through it. The throughput rate of a pipeline is a measure of how many data values can pass through it per unit time.

Pipelines both store and process data; the storage elements and processing logic in them alternate along their length. I will describe pipelines in their complete form later, but first I will focus on their storage elements alone, stripping away all processing logic. Stripped of all processing logic, any pipeline acts like a series of storage elements through which data can pass.

Pipelines can be clocked or event-driven, depending on whether their parts act in response to some widely-distributed external clock, or act independently whenever local events permit. Some pipelines are inelastic; the amount of data in them is fixed. The input rate and the output rate of an inelastic pipeline must match exactly. Stripped of any processing logic, an inelastic pipeline acts like a shift register. Other pipelines are elastic; the amount of data in them may vary. The input rate and the output rate of an elastic pipeline may differ momentarily because of internal buffering. Stripped of all processing logic, an elastic pipeline becomes a flow-through first-in-first-out memory, or FIFO. FIFOs may be clocked or event-driven; their important property is

that they are elastic.

I assign the name micropipeline to a particularly simple form of event-driven elastic pipeline with or without internal processing. The micro part of this name seems appropriate to me because micropipelines contain very simple circuitry, because micropipelines are useful in very short lengths, and because micropipelines are suitable for layout in microelectronic form.

I have chosen micropipelines as the subject of this lecture for three reasons. First, micropipelines are simple and easy to understand. I believe that simple ideas are best, and I find beauty in the simplicity and symmetry of micropipelines. Second, I see confusion surrounding the design of FIFOs. I offer this description of micropipelines in the hope of reducing some of that confusion.

The third reason I have chosen my subject addresses the limitations imposed on us by the clocked-logic conceptual framework now commonly used in the design of digital systems. I believe that this conceptual framework or mind set masks simple and useful structures like micropipelines from our thoughts, structures that are easy to design and apply given a different conceptual framework. Because micropipelines are event-driven, their simplicity is not available within the clocked-logic conceptual framework. I offer this description of micropipelines in the hope of focusing attention on an alternative transition-signalling conceptual framework.

We need a new conceptual framework because the complexity of VLSI technology has now reached the point where design time and design cost often exceed fabrication time and fabrication cost. Moreover, most systems designed today are monolithic and resist mid-life improvement. The transition-signalling conceptual framework offers the opportunity to build up complex systems by hierarchical composition from simpler pieces. The resulting systems are easily modified. I believe that the transition-signalling conceptual framework has much to offer in reducing the design time and cost of complex systems and increasing their useful lifetime. I offer this description of micropipelines as an example of the transition-signalling conceptual framework.

Until recently only a hardy few used the transition-signalling conceptual framework for design because it was too hard. It was nearly impossible to design the small circuits of 10 to 100 transistors that form the elemental building blocks from which complex systems are composed. Moreover, it was difficult to prove anything about the resulting compositions. In the past five years, however, much progress has been made on both fronts. Charles Molnar and his colleagues at Washington University have developed a simple way to design the small basic building blocks [9]. Martin Rem's "VLSI Club" at the Technical University of Eindhoven has been working effectively on the mathematics of event-driven systems [6, 10, 11, 19]. These emerging conceptual tools now make transition signalling a lively candidate for widespread use.

TWO CONCEPTUAL FRAMEWORKS

In the clocked-logic conceptual framework, registers of flip flops operating from a common clock separate stages of processing logic. Each time the clock enters its active state a new data element enters each register. Data elements march forward through successive registers in lock step, each taking a fixed number of clock cycles to pass through the fixed number of registers and intervening logic stages built into the system. The clocked-logic conceptual framework is widely used 1) because it offers a simple way to design computing equipment, 2) because it is widely taught and understood, 3) because parts that operate with clocks are widely available, and 4) because system noise has died away by the time a clock event occurs.

To build the micropipelines described here we must discard the clocked-logic conceptual framework and think instead about a different but equally simple form

of control called transition signalling. In return for moving into the transition-signalling conceptual framework, we are rewarded with three new types of flexibility. In hardware design we attain a new flexibility to compose systems from small parts previously designed and tested; in software, we achieve a new flexibility to handle vectors of variable length; and in systems we enjoy a new flexibility to extend system life by replacing isolated parts whenever components with improved speed or cost become available.

It is often hard to discard a conceptual framework. The well-known puzzle shown in Figure 1 illustrates this difficulty by asking us to draw four straight lines through nine dots without lifting our pencil from the paper. Our natural conception of figure and ground in looking at this puzzle suggests that the lines to be drawn should stay within the square of dots, a conceptual framework that renders the task impossible. The puzzle can be solved only by drawing outside the boundary of the dots.

Similar difficulty in discarding a conceptual framework can be seen in the design of FIFOs. Conventional wisdom in the clocked-logic conceptual framework says that each stage of a flow-through FIFO should have a clocked register that feeds its output to the input of the next stage. Now recall that a FIFO must be elastic; it must be able to store a variable amount of data; and if it has a fixed number of stages, some of them may be unoccupied. Continuing with the clocked-logic conceptual framework, a "full" or "empty" clocked flip flop for each stage is required to make the FIFO elastic.

Each stage must also have logic involving the state of its full or empty flip flop and the states of other stages to decide when to capture new data. One simple control rule operates as follows: a) Each stage captures new data and sets its full flip flop to the full state whenever

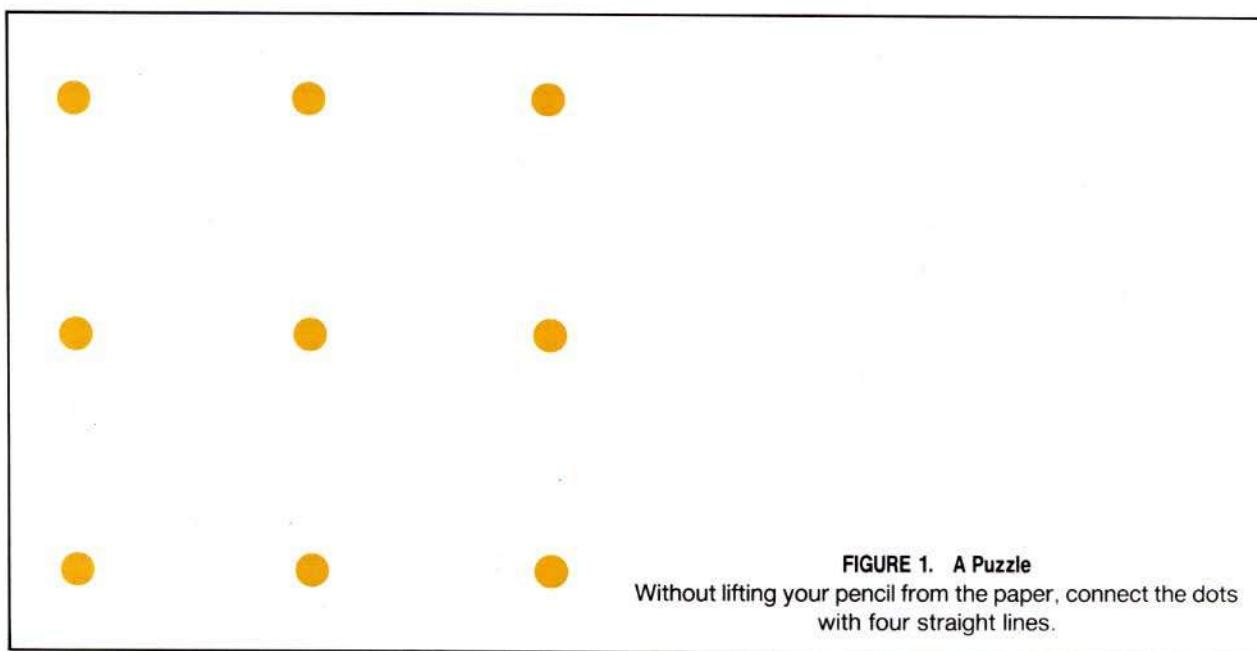


FIGURE 1. A Puzzle

Without lifting your pencil from the paper, connect the dots with four straight lines.

it is empty and its predecessor is full. b) Each stage sets its full flip flop to the empty state whenever it is full and its successor is empty. This rule delivers output data only on alternate clock cycles. More complex rules for the control of synchronous FIFOs get better performance by looking ahead many stages to decide if an entire block of data can move forward during the forthcoming cycle. The clocked-logic conceptual framework itself creates this complexity, because all registers must act together at once; any that fail to act now must suffer a complete cycle of delay for their next opportunity.

The clocked-logic conceptual framework is poorly matched to FIFO design for another reason as well: FIFOs often connect senders and receivers that have separate clocks. The difficulty of designing a FIFO with separate clocks at input and output is strikingly evident when one asks whether to use the input or the output clock to control the internal stages. There is no natural point in the FIFO to transfer control from the input clock to the output clock. Should the transfer occur near the beginning, at the middle, or near the end of the pipeline? Why?

If conventional clocks are used at the input and output of a FIFO and the two clocks are separate, then arbitration or synchronization between these two clocks is required somewhere in the design. Arbitration or synchronization is necessary because the data must pass from control by the input clock to control by the output clock somewhere, even though the phase relationship between the clocks is unknown and variable. There is always some phase relationship between the separate clocks that violates the setup or hold time requirements of some latch or flip flop.

The fact that arbitration or synchronization is required somewhere in a clocked FIFO introduces a host of problems [1]. It is not possible to make an arbiter or synchronizer that is perfectly reliable; instead one must design a circuit for which the probability of failure is so low as to be unimportant. Sadly, although the problems inherent in synchronizers and arbiters have been known for many years, synchronizer failures still cause difficulty, and remarkably, discussions of arbitration and synchronization are largely absent from the descriptions of FIFOs now on the market. Solutions to the inherent synchronizer problems are left to the users.

The internal stages of the micropipelines I shall describe here get their timing signals neither directly from the input control signals nor directly from the output control signals. Rather, each internal stage captures a new data value whenever its successor stage has accepted the present value and its predecessor stage has the new data ready. Each stage operates at its own pace, using control information only from adjacent stages. Letting each stage operate separately, without a common clock, avoids the need for arbitration and simplifies the design. Although the design of FIFOs and other micropipelines is very difficult within the clocked-logic conceptual framework, it is easy once one abandons that framework in favor of transition signalling, as I shall do throughout this lecture.

TRANSITION SIGNALLING

In transition signalling any transition, either rising or falling, has the same meaning, as illustrated in Figure 2; either kind of transition is called an event. As indicated in the figure, and suggested by its name, transition signalling avoids distinguishing between the two types of transitions even though they may look quite different. In effect, all responses to transition signals are edge-triggered, and are triggered on both rising and falling edges. Because transition signalling uses both rising and falling edges as trigger events, it may offer twice the

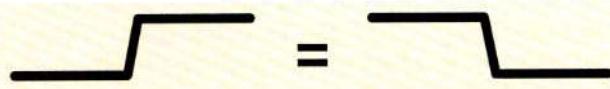


FIGURE 2. Two Equivalent Transistors

Rising and falling transitions on signalling wires have the same meaning. They are called events.

speed potential of conventional clocking.

Transition signalling avoids assigning meanings to the absolute high or low state of control signals. I will use the state of a control signal only relative to the states of other related control signals; the state of a control signal may be the same as or different from that of another, but its absolute state will never matter. Because the absolute state of a transition control signal is unimportant, there is no need to return it to some neutral state between events. By avoiding such returns to a neutral or low state, transition signalling saves the time and energy costs of the return transition, as well as the design confusion of an unnecessary event. Transition signalling is much like non-return-to-zero (NRZ) magnetic recording.

Many people find it difficult at first to grasp the notion that both rising and falling edges should have the same meaning. This is not surprising because a change in conceptual framework is required. Most people are accustomed to differentiating high and low levels and to a clock that returns to a neutral state between actions. But even though it may seem hard at first, the transition-signalling conceptual framework quickly becomes very easy to use. Abandoning the clocked-logic conceptual framework in favor of the transition-signalling conceptual framework can provide real advantages in speed and simplicity that are particularly striking in micropipelines.

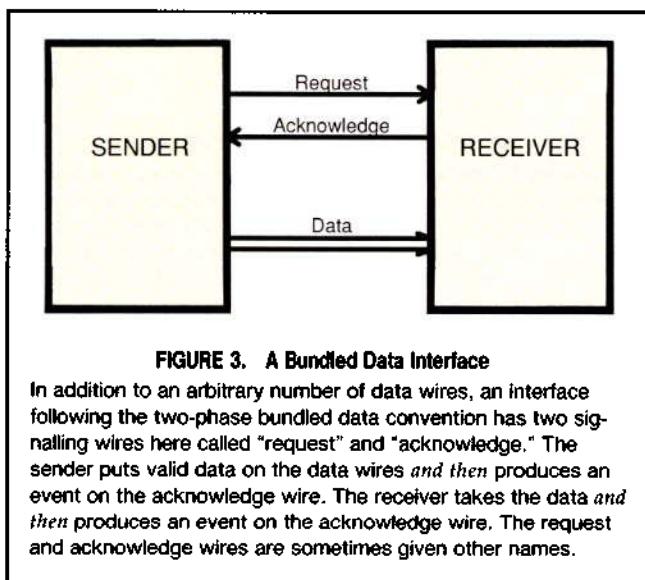
Transition signalling circuits must be symmetric with respect to the high and low states of control signals, since both rising and falling transitions have the same meaning. Look for this symmetry throughout this lecture. Notice also that my descriptions of circuit action speak of control signal levels only in relative terms as being the same or different rather than in absolute terms as being high or low, again to preserve symmetry. I use conventional levels, high or low, only for data values. The symmetry of transition signalling provides appealing simplicity in complementary metal oxide

semiconductor (CMOS) circuits because it fits well with the symmetry of the complementary transistors from which CMOS circuits are built.

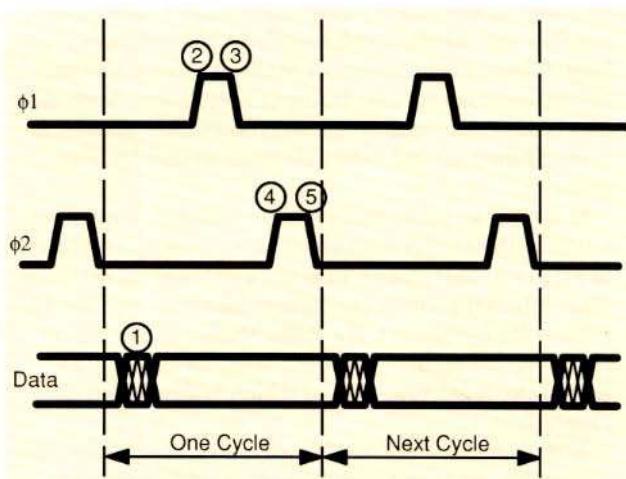
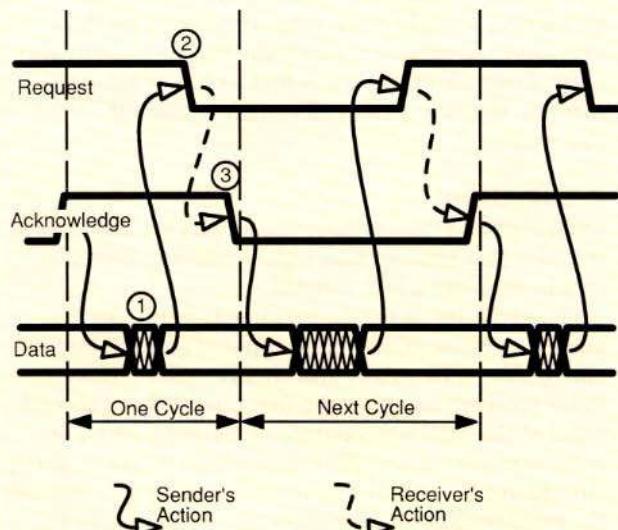
THE TWO-PHASE BUNDLED DATA CONVENTION

If a sender and a receiver communicate using transition signalling, there will be two control wires and many data wires between them, as illustrated in Figure 3. The data wires carry conventional high or low states to convey true or false Boolean data. The sender places a data value on the data wires *and then* produces an event on its control wire, called "request," to indicate that valid data are available. In some cycles the request event will be a rising transition and in some it will be a falling transition; we make no distinction between them. The receiver accepts the data *and then* produces an event on its control wire, called "acknowledge," to indicate that the data have been accepted. The three events, data change, request event, acknowledge event, always follow in cyclic order, as illustrated in Figure 4. Successive cycles may take different amounts of time, as suggested by the difference in length of the cycles in the figure. Sometimes names other than "request" and "acknowledge" are used for the two control wires [7]. You may wish to compare this protocol to the non-overlapping clock protocol of Figure 5.

Seitz [13] describes the protocol of Figure 4 that we have come to call the two-phase bundled data convention. The "two-phase" part of this name indicates that only two phases of operation are distinguished: the sender's active phase and the receiver's active phase. An event terminates each phase: the request event terminates the sender's active phase, and the acknowledge event terminates the receiver's active phase. The sender is free to change the data during its active phase and makes an event on the request wire after it has made the data valid; it must then hold the data constant during the receiver's active phase. The "bundled data" part of this name indicates that the data wires



and the request signalling wire must be treated as a bundle; delays in data transmission must be less than delays in transmitting the request event lest the request event reach the receiver prior to valid data. The ac-



knowledge wire need not be bundled.

In addition to several data wires, an interface using the two-phase bundled data convention requires two control wires. One may think that this is more expensive than a conventional clocking system, which requires only a single clock wire. The two wires, however, serve to replace not only the clock wire, but also at least two additional wires that would be required between stages in a clocked system to make the pipeline elastic.

EVENT LOGIC

Control circuits for transition signalling are built out of modules that form various logical combinations of events. Here are a few samples:

The exclusive or (XOR) circuit acts as the OR element for events. When either input of an XOR circuit changes state, its output also changes state. Thus an event received on either the first input OR the second input of the XOR will produce an output event. For more than two inputs, XOR generalizes to parity; parity circuits act as the multiple-input OR for events. We use the standard XOR logic symbol with two or more inputs to represent these OR elements for events. Such elements are sometimes called MERGE elements, because they merge two or more event streams.

The Muller C-element [8], for which I will shortly show circuits, acts as the AND element for events. When both inputs of a Muller C-element are in the same logical state, the Muller C-element's state and its output are copies of that state. When the two inputs differ, the Muller C-element uses internal storage to retain its previous state and hold its output unchanged. Thus only after an event takes place on both of its inputs will a Muller C-element produce an event at its output. The Muller C-element generalizes easily to three or more inputs, requiring that all of them reach a new logical state before copying that new state as output. We use the standard AND logic symbol with a large C inside it to represent Muller C-elements that implement logical AND for transition events. Such elements are sometimes called RENDEZVOUS elements, because they act only after all input events have arrived.

In CMOS an appealingly simple dynamic implementation of the Muller C-element is possible, as illustrated in Figure 6. This circuit uses the electrical capacitance of an internal node as the storage element required in the Muller C-element. If a static Muller C-element is required, the node capacitance must be augmented with static logic, one form of which is illustrated in Figure 7.

Although the absolute state of a transition signal does not matter, its state relative to other related signals does matter. Thus it is sometimes important to invert transition signals. We use "bubbles" on inputs or outputs of logic symbols to represent such inversions, as illustrated in Figure 8. Every loop around which events flow must contain an odd number of inversions. Such loops are, in effect, oscillators whose oscillations are

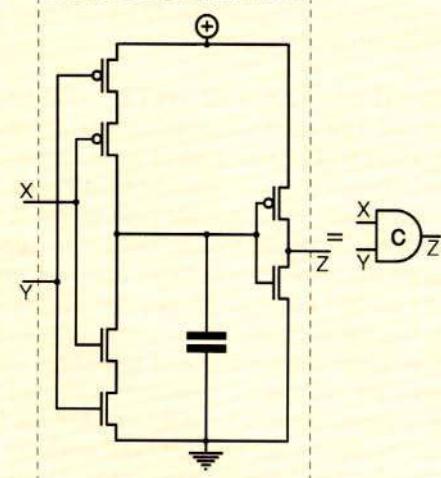


FIGURE 6. A Dynamic Muller C-Element

In CMOS the Muller C-element has a particularly simple dynamic implementation that uses the electrical capacitance of an internal node as the storage element. Transistors to initialize the Muller C-element during master clear are not shown.

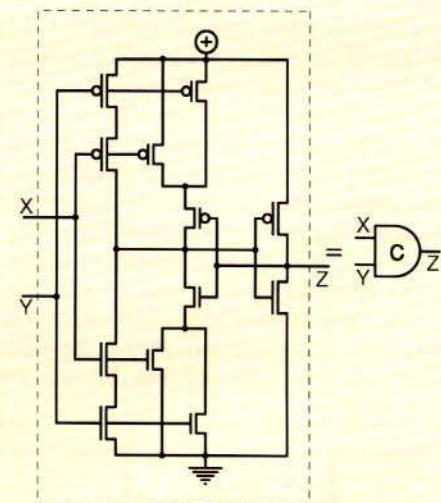


FIGURE 7. A Static Muller C-Element

Replacing the capacitor of Figure 6 with transistors as shown produces a static Muller C-element. In an integrated circuit layout, the transistors shown here with smaller symbols can be very narrow, since they serve only to retain an already-established value. This circuit generalizes in an obvious way to three or more inputs.

coordinated with those of other loops by the actions of Muller C-elements or other modules at loop junctions.

Figure 9 shows block symbols for some other useful event logic circuits that implement elemental operations, some of which are familiar to programmers. The TOGGLE circuit produces events alternately on its two outputs in response to events at its input; the first event after some master clear signal and every other subse-

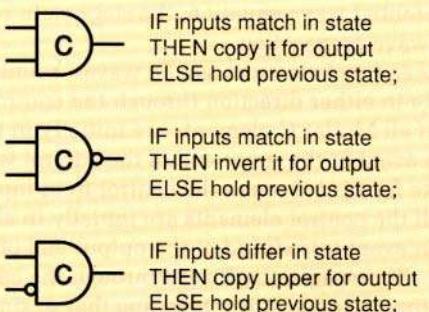


FIGURE 8. Muller C-Elements with Inverters

Muller C-elements contain storage to hold a previous state on some input conditions. When inverters are included in input or output wires, as indicated by the bubbles in this figure, the actions are as listed. Muller C-elements provide the AND function for events.

quent event pass through it to the output with the dot. The SELECT module steers an incoming event to one output or the other depending on the value of a data input; it serves for testing the Boolean condition in conditional expressions. The Boolean value must be available before the incoming event that it steers, a requirement similar to the bundling condition in the protocol of Figure 4. The CALL element remembers which of its inputs most recently received an event, and returns an event to the matching output terminal after a called procedure has finished. The memory in the CALL element serves the role of subroutine return address. The CALL element operates properly only if each call completes before a subsequent call occurs. The ARBITER decides cleanly between two events whose arrival sequence is unknown, producing a grant event for only one of them even if they arrive at very nearly the same time. Like a semaphore in programming, it delays subsequent grants until after receiving an event on the done wire corresponding to an earlier grant so that only one grant at a time is ever outstanding. An ARBITER can be connected directly to a CALL element to permit two entirely independent processes to call on a single shared procedure.

CONTROL FOR MICROPIPELINES

A string of Muller C-elements with inverters interposed, as illustrated in Figure 10, is the only logic required to control the micropipelines described in this lecture. I find it remarkable that the only distinctions between the forward and reverse direction of the pipeline to be found in this circuit are the delay required for bundling and an inversion in the reverse signal path. Observe in Figure 10 that a request and an acknowledge signal pass between adjacent stages of this control. Data wires also pass between stages, as I shall shortly describe, but these are not shown in Figure 10. At each interface between stages the request and acknowledge signals and the data values follow exactly the two-phase bundled data convention of Figure 4.

Notice also in Figure 10 that the request and ac-

knowledge wires at the input interface are identical to those at the output interface. The similarity of the signalling form at the input and output ends of the control ensure that any number of such control systems, even ones that differ markedly in raw speed, will operate properly when connected in series, albeit at the speed of the slowest. This composability of micropipelines makes it easy to assemble long signal-processing pipelines; one simply connects the request, acknowledge, and data wires at the output of one micropipeline to the corresponding wires at the input of the subsequent micropipeline. The composite control is just a further repetition of the control system illustrated in Figure 10.

One way to see how the control circuit of Figure 10

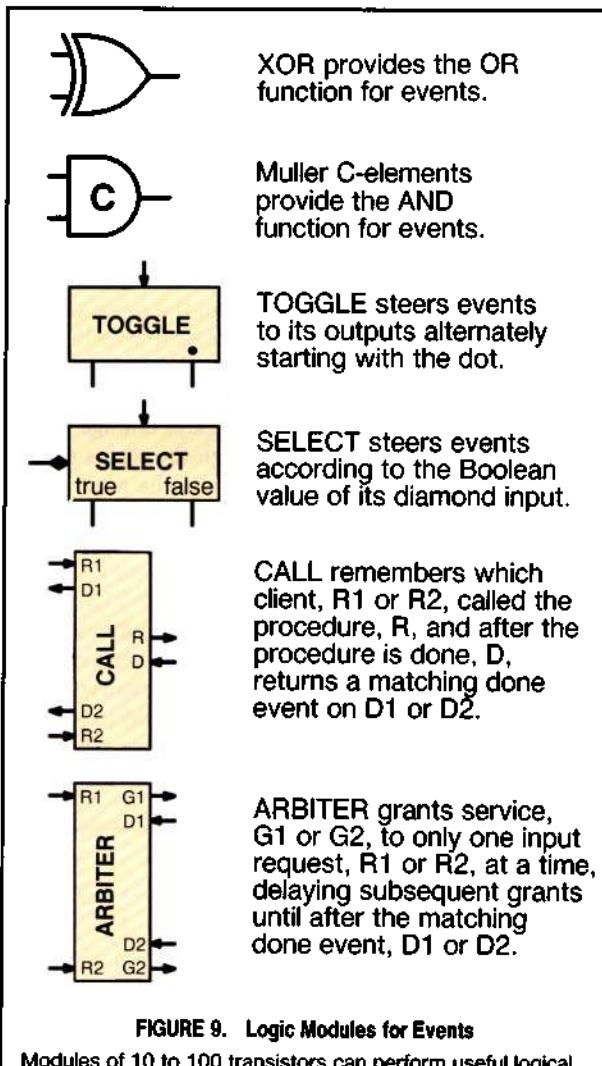


FIGURE 9. Logic Modules for Events

Modules of 10 to 100 transistors can perform useful logical functions on events. The modules whose symbols are shown provide the functions indicated. Note the similarity of these functions to the basic structures used in programming. One might think that the arbiter would require 8 terminals, since the request signals at the left seem to lack corresponding acknowledge signals. Either the grant or the done signals are used to acknowledge incoming requests, depending on the application.

works requires us to focus on its behavior as a series of loops around which events flow. There is a single inverter in each loop, and so each loop will oscillate. The Muller C-elements coordinate the oscillations in adjacent loops. This view makes it easy to see that the request and acknowledge events at each interface must alternate.

Another way to see how this circuit works requires us to focus on the state of each Muller C-element relative to the states of predecessor and successor Muller C-elements. Remembering the behavior of a Muller C-element with one inverted input, we can easily see that each stage of the control of Figure 10 follows a very simple stage state rule:

IF predecessor and successor differ in state
THEN copy predecessor's state
ELSE hold present state.

This stage state rule makes the control system stable both when all stages are in the same state and when alternate stages are in opposite states. The condition in which all control elements are in the same state corresponds to an empty pipeline, and the condition in which alternate stages are in opposite states corresponds to a filled pipeline. There are other stable conditions with stages near the input end in the same state and stages near the output end in alternating states; these conditions correspond to a partly filled pipeline. The stage state rule also makes the control system unstable in some states; such unstable states change immediately as events propagate through the stages of the pipeline. To initialize a micropipeline to the empty condition, its Muller C-elements may all be set to the same state by a master clear signal. I have omitted the reset circuits required to do this from Figures 6 and 7.

The stage state rule, described above in IF THEN ELSE form, is the digital equivalent of the differential equations that describe ocean waves and electromagnetic waves. In a wave equation a time derivative, in this case copy predecessor's state, is set equal to a space derivative, in this case IF predecessor and successor differ in state. Like the rules of physics described by

the differential wave equation, the stage state rule results in wave propagation.

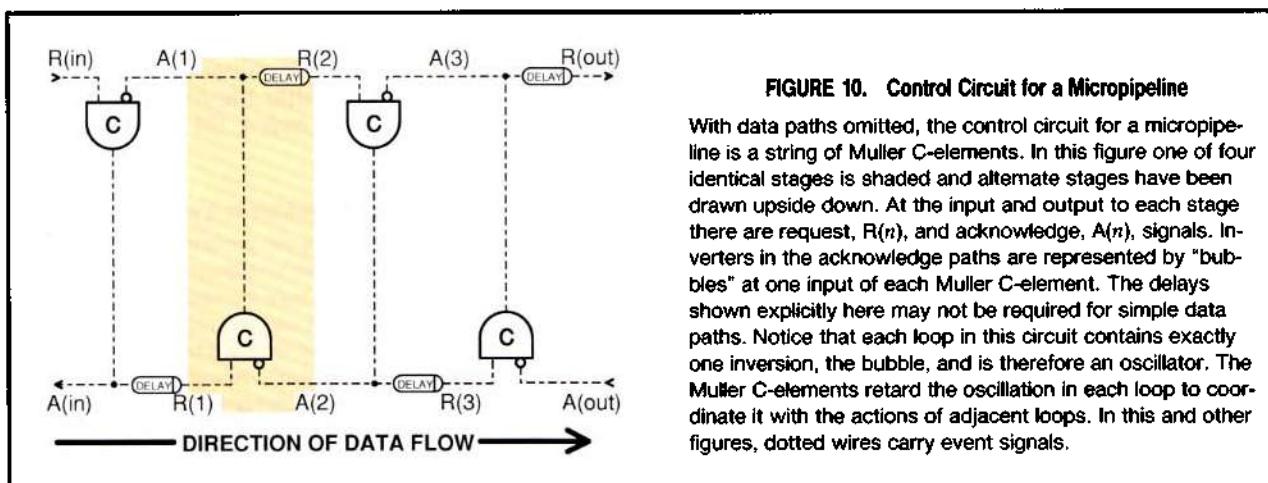
Like ocean and electromagnetic waves, events can propagate in either direction through the control of Figure 10. If all Muller C-elements are initially in the same state, an event at the input end of the control will propagate forward through the control from input to output. If the control elements are initially in alternate states, an event introduced at the output end of the control will propagate backward through the control from output to input. It is interesting that so simple a circuit should exhibit wave propagation in both directions.

Both forward and reverse propagation of events in the control system of Figure 10 are useful in controlling micropipelines. Forward propagation of events through the control circuit will force information forward through the micropipeline, much as an ocean wave pushes a surfer toward the shore. Reverse propagation will sweep empty data spaces created at the end of the pipeline back through occupied sections toward the beginning. Empty spaces move through occupied sections much as holes move in a semiconductor or air bubbles rise through water.

AN EVENT-CONTROLLED STORAGE ELEMENT

This section introduces a storage element suitable for use with a transition signalling control system. In order to make these circuits easier to understand, I will use a switch symbol to represent any one of several configurations of transistors, one of which is shown in Figure 11. As you can see in the figure, the transistor implementation of this switch makes use of both the true and the complement form of its control signal, C and $\sim C$, which implies an inversion of the control signal not shown explicitly in the figure. Notice that the circuit is entirely symmetric with respect to high or low values of its control signal, selecting one input when its control is high and the other when its control is low. I will always draw such switches in the position they assume when their control signals are low.

A suitable storage element for use with a transition



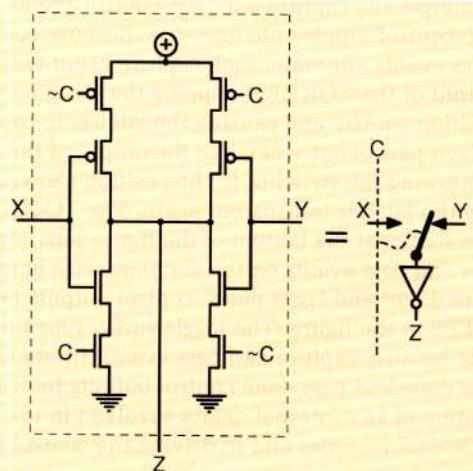


FIGURE 11. Circuit for the Switch Symbol

The double-throw switch symbol at the right of this drawing represents the transistor circuit shown inside the dotted line. When the control wire, C , is low, the output terminal, Z , is controlled by the Y input, as shown. When the control wire is high, the switch flips to the X input. The output of this form of switch is controlled by its selected input, but inverted in value. Other implementations of such a switch using pass transistors are also possible.

signalling control system must respond to transition events. Unlike a conventional latch in which the "high" and the "low" state of the clock signals can perform different functions, an event-controlled storage element must give similar responses to rising and falling transitions. Suitable circuits that use two control wires called "capture" and "pass" are illustrated in Figure 12. Each of these two circuits uses two latches side by side and

activates them alternately. In the circuit with only three inverters, the output inverter is shared between the two latches. Notice that because of the inverters implied in the control of the switches shown, both of these circuits are entirely symmetric with respect to high and low values of their control signals. You may wish to compare these circuits with the conventional D flip-flop circuit shown in Figure 13.

The behavior of an event-controlled storage element is easy to describe using only the relative states of its two control signals. When its two control signals are in the same state, the condition shown in Figure 12, the event-controlled storage element is transparent and delivers its input data directly to its output, not acting as a storage element at all. You can see a path through the switches and inverters leading directly from input to output. When its two control signals differ in state, one or the other of the switch sets will be flipped from the position shown in Figure 12. As you can imagine from the figure, if one of the switches is flipped, a loop is formed containing two inverters. Such a loop captures and retains the data value. If one switch is flipped to form such a loop, no path exists from input to output, the event-controlled storage element is rendered insensitive to changes on its data input terminal, and it reports at its output only the data captured in the loop.

The behavior of an event-controlled storage element can also be described in terms of events. Let us assume that the event-controlled storage element is initially transparent, as it is shown in Figure 12 and that the capture and pass control signal events always alternate. An event on its capture control wire flips the two switches to which the capture wire is connected, and thus causes the storage element to capture and hold the data value then passing through it. This event isolates the output value of the element from changes at the

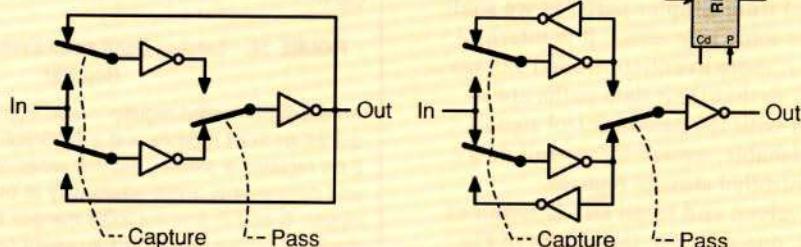


FIGURE 12. Event-Controlled Storage Elements

An event-controlled storage element responds to events on its two control wires, called "capture" and "pass" in this drawing. Two different configurations are shown. The form on the right, with five inverters, is slightly faster than the form on the left, with only three inverters, because its feedback paths contain only one switch rather than two. After master clear the switches will be in the position shown, making a direct connection without loops between input and output, a state in which

the storage element is said to be transparent. Storage elements of either type are formed into registers just as are flip flops by connecting their capture and pass control wires in parallel. The register symbol includes control outputs, Cd and Pd , which are amplified, and thus necessarily delayed, versions of the control input signals, C and P . Cd and Pd , named for "capture done" and "pass done," deliver output events after the register has done its action.

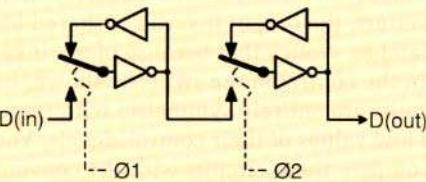


FIGURE 13. Conventional D Flip-Flop

A conventional D flip flop is controlled by non-overlapping clock signals ϕ_1 and ϕ_2 illustrated in Figure 5. Compare this circuit to the event-controlled storage element of Figure 12.

element's input but does not change the output value. A subsequent event on the pass control wire flips the other switch, returning the element to the transparent state, permitting the next data value to appear as its output, and possibly changing its output value. After each event on the element's pass control wire a new output value appears. This is exactly the behavior required to make micropipelines with the control system already described.

Event-controlled storage elements are connected in groups to form event-controlled registers. Each such register consists of a number of event-controlled storage elements with their capture and pass wires connected in parallel. Because the capture and pass wires drive many transistors, suitable amplifiers are included in the register design. These amplifiers have some unknown delay, and there is further delay introduced by the physical length of the wires. I therefore include pass done, Pd, and capture done, Cd, event outputs on every event-controlled register, as shown on the register symbol in Figure 12. Events on these outputs follow exactly the events on the capture and pass control inputs, but are delayed to account for the amplification and wiring delays in the register. The explicit done signal outputs permit subsequent actions to be further delayed if required. Such a delay might be needed if the register is composed from simpler parts, as we shall shortly see, or performs some side effect. It is interesting to note that if two or more event-controlled storage registers are connected so that their data paths are in series and are provided with the same control signals, the result is indistinguishable, except for overall delay, from a single event-controlled storage register.

If wide words are involved and small size is required rather than high speed, one may use the circuit of Figure 14 as an event-controlled storage register. This circuit uses only a single latch per bit, but requires extra equipment for control. The extra control equipment is required because the latches have only a single control wire in which both capture and pass events must flow. Naturally, the delays introduced by the extra equipment also delay the capture done and pass done control outputs.

The operation of the circuit of Figure 14 is easy to understand. The XOR circuit shown at the top of the

figure merges the capture and pass control events from the two control inputs onto one wire. Because capture and pass events alternate, each capture event will make the output of the XOR high, flipping the switches from the position shown, and causing the latches to capture data. Each pass event will make the output of the XOR low, returning the switches to the position shown, and making the latches transparent again. The TOGGLE module shown at the bottom of the figure separates the capture and pass events on the common wire onto the "capture done" and "pass done" control outputs labeled Cd and Pd in the figure. The toggle suffices for this purpose because capture and pass events alternate. The capture done and pass done control outputs indicate completion of any internal delays involved in the XOR and TOGGLE modules and in driving the control wire for the many latches in the register.

MICROPIPELINES WITHOUT PROCESSING

A micropipeline with no processing in it, which is a FIFO, can be built by combining the control of Figure 10 with the storage registers of Figure 12 or Figure 14. A set of event-controlled storage registers in series

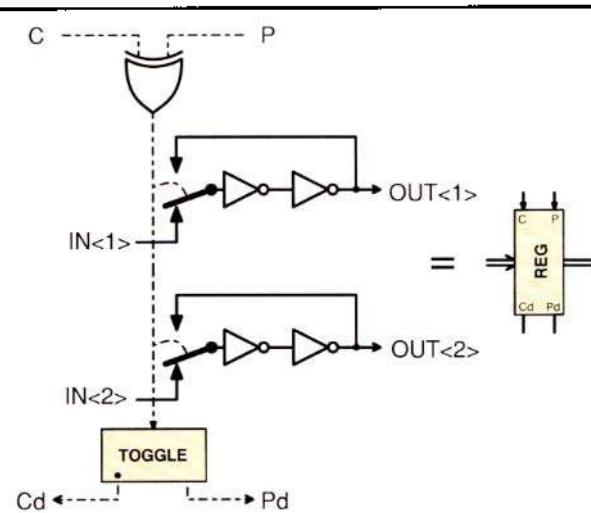


FIGURE 14. Latches Used as an Event-Controlled Storage Register

An event-controlled register made from ordinary latches requires an XOR module and a TOGGLE module for control. A 2-bit register is shown; dashed wires carry events. Capture and pass events arrive alternately at the separate control inputs, C and P, but the XOR merges them onto one wire. At the XOR output, each capture event becomes a rising transition in the latch control wire and flips the switches, causing the latches to capture data. Each pass event becomes a falling transition in the latch control wire and flips the switches back to the position shown, making the latches transparent again. The TOGGLE module separates the capture and pass events back into two separate output paths, Cd and Pd, after the register has done its action. This circuit is slower than the event-controlled register of Figure 12 and delays its output events, Cd and Pd, accordingly, but except for delay provides exactly the same function.

serves as its data path while a string of Muller C-elements serves as its control, as illustrated in Figure 15. Each event-controlled storage register uses the control signal from its stage of the control as its capture control signal, and the control signal from the successor stage as its pass control signal. When this FIFO is empty, all of its storage registers are transparent, and so a path exists through it directly from its data input terminals to its data output terminals.

I have arranged the layout of Figure 15 not only to make it easy to read but also to suggest a layout for an integrated circuit implementation. The Muller C-elements are located at either ends of the registers, just as shown, so that control signals zigzag across the chip. The wires that control the registers are driven from one side of the register and are used to control the Muller C-elements of adjacent stages at the other side of the register. Because the control signals for the register must be amplified to drive all the switches in the many storage elements involved, and because the wires that carry control signals across the register are long, there is always some delay in controlling the register. The arrangement of driving registers from one side and sensing their control signals at the other side accommodates not only the delay in the driving amplifiers but also any delay in the wires themselves.

If no processing is required in the micropipeline, i.e., for a FIFO, the simpler data path circuit of Figure 16 will serve [17]. In this circuit the side-by-side latch configuration of the event-controlled storage element is extended between stages. The two separate data paths are brought together again only at the output end of the FIFO by an output selector switch very similar to that used in the event-controlled storage element. The first, third and other odd-indexed data values pass through the upper data path while even-indexed values pass through the lower data path.

Look at all the symmetry in Figure 16. Except at its output, shown at the right of the figure, there is no distinction at all in its data path between the forward and reverse directions of the FIFO. It seemed at first to me that this must indicate a flaw in its design. There is no flaw; the circuit of Figure 16, though unconventional, works well. Abandoning the conventional notion

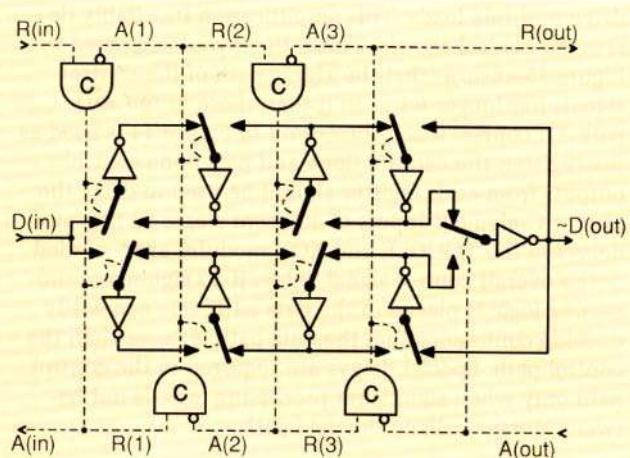


FIGURE 16. A FIFO Circuit

If no processing is required, as in a FIFO, the event-controlled storage elements in Figure 15 can be replaced with this simpler circuit [17]. In this figure, dashed lines carry control signals, solid lines carry data values, and four stages are shown. The switches are drawn as they would be in an empty FIFO. The FIFO illustrated is one bit wide and can therefore store four bits; more length or width comes with further repetition of the internal parts of this data path. Alternate inputs pass through the upper and lower rails of the data path and merge again only at the output. When the FIFO is empty, as it is illustrated, it is transparent; one can trace a direct path from data input to data output. When each switch changes, identical data is presented to each of its inputs; thus the switches may momentarily short their two inputs together when changing. The micropipeline shown has an odd number of inversions and thus inverts its data value.

of a latch produces a simple and effective circuit for a FIFO.

Naturally, data must propagate through a micropipeline faster than the control events propagate through its control. This is usually assured by three factors. First, the Muller C-element used in the control circuit is more complex than the storage element used in the data path and therefore inherently slower. Second, since each single stage of the control system must drive the many storage elements that hold a parallel word in each register, the control signals must be amplified to

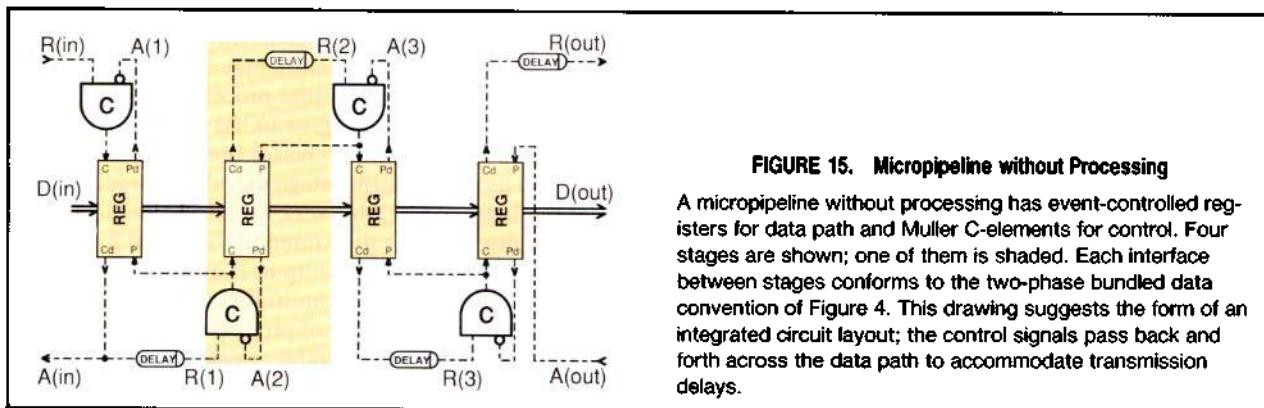


FIGURE 15. Micropipeline without Processing

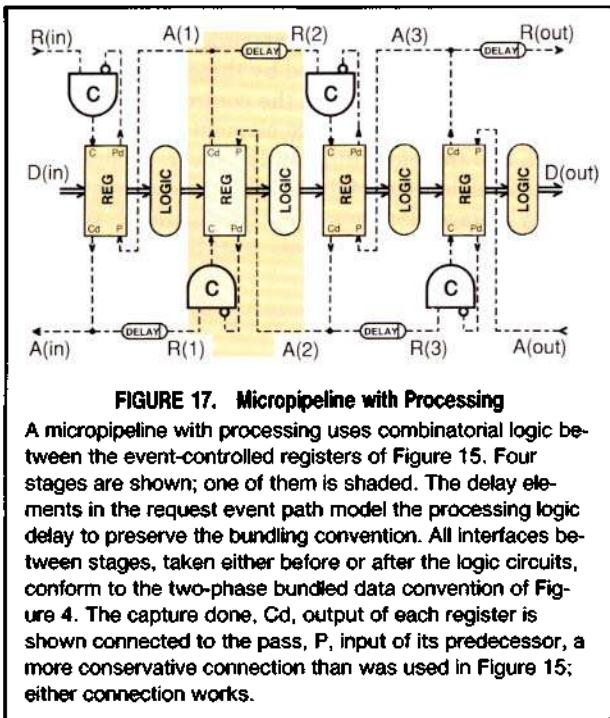
A micropipeline without processing has event-controlled registers for data path and Muller C-elements for control. Four stages are shown; one of them is shaded. Each interface between stages conforms to the two-phase bundled data convention of Figure 4. This drawing suggests the form of an integrated circuit layout; the control signals pass back and forth across the data path to accommodate transmission delays.

drive multiple loads. This amplification inevitably delays the control signals. Third, the layout suggested in Figure 15 ensures that the zigzag path of the control signals has longer wires in it than those in the data path. Of course, when the circuit of Figure 14 is used as the register, the capture done and pass done control outputs from each register should be used to drive the Muller C-element inputs of adjacent stages so that any delays in the TOGGLE and XOR modules are included in the overall control signal delay. If no significant processing logic is placed in the data path, one can easily develop confidence that the data path is faster than the control path. Special delays are required in the control path only when significant processing logic is put between storage cells in the data path.

MICROPIPELINES WITH PROCESSING

The micropipeline framework provides a basis for a variety of pipeline processors [18]. My colleagues and I have designed multipliers, binary to one-out-of-N decoders, a memory controller, and other circuits using the micropipeline framework. In each case the micropipeline control template of Figure 10 provides the basis for the design. In some cases this simple template is embellished with circuits composed from the event logic modules of Figure 9 to provide more complex control functions. For example, using only one TOGGLE module and one XOR module it is easy to construct a circuit that performs two operations for each input event. The same two modules connected differently form a circuit that performs an operation only for every other input event.

When logical processing is required, suitable combinatorial circuits are placed between the storage regis-



ters, as illustrated in block form in Figure 17. One can trade off the number of stages of storage and the complexity of the intervening logic to obtain a suitable balance between latency and throughput rate. With less combinatorial logic between stages and more stages of storage, one obtains higher throughput rate at the cost of greater latency. The decoder circuits of Figures 18 and 19 perform the same function using different amounts of storage.

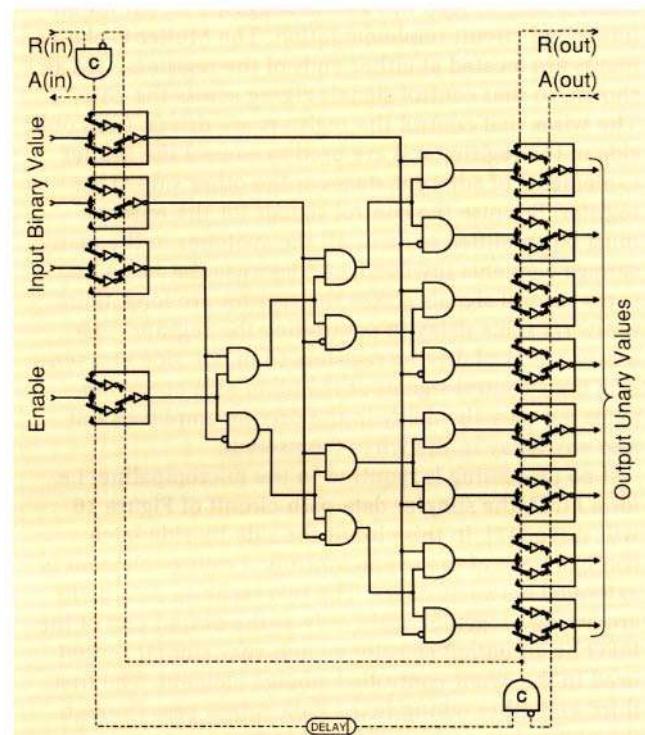


FIGURE 18. A Decoder with Two Micropipeline Stages

This two stage micropipeline decodes three binary input bits into eight unary output bits. It can store two values, one not yet decoded and the other fully decoded. The processing logic at the center of the figure is formed from three ranks of ordinary AND gates. The delay at the bottom of the figure must delay the request event at least as much as the three ranks of combinatorial logic delay the data.

The number of bits of storage in the registers of successive stages in a micropipeline may vary widely according to the needs of different processing steps. For example, the decoder of Figure 19 has 3 binary inputs but 8 unary outputs, and increases the width of the data word as each internal stage decodes an additional bit of the input. The 12-bit \times 12-bit micropipeline multiplier whose layout is illustrated in Figure 20 has 24-bit data paths at input and output. It uses 24 stages of micropipeline: 12 to do the multiplication and 12 to resolve the carry-save form of product that results. At the center of the pipeline, 36-bit registers are required, since half of the product is in a carry-save form that requires two bits of data to represent each bit of product.

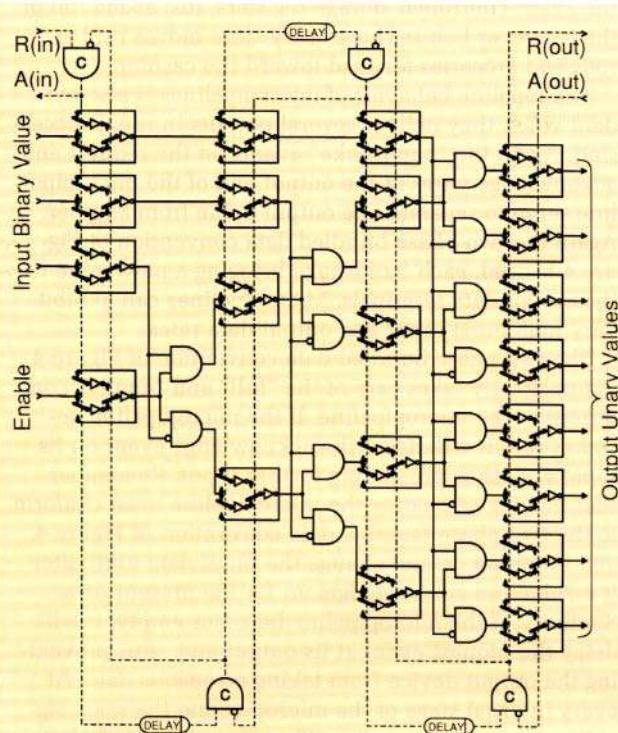


FIGURE 19. A Decoder with Four Micropipeline Stages

This four stage micropipeline also decodes three binary input bits into eight unary output bits. It does the same thing as the circuit of Figure 18, but it has more storage for partially decoded results. It can store four values, the first not yet decoded, two partially decoded values, and the final one fully decoded. The three delays in these request paths can each be shorter than the one in Figure 18, because each delay models only a single rank of combinatorial logic. This decoder has a higher throughput rate than the functionally equivalent decoder of Figure 18.

Because it has 24 stages, this multiplier can hold as many as 24 partially processed products. It can also hold fewer. It automatically processes any partially complete products as much and as fast as possible, considering the products already queued for output. At full operating speed, it provides the very high throughput of a pipeline process. When empty, however, it has no storage and acts as a combinatorial multiplier to produce individual products. It is never necessary to insert dummy data to flush previously entered information out of a micropipeline.

As a more complex example, we designed a memory controller using the micropipeline framework. This memory controller is intended for byte-serial access to a dynamic random access memory (DRAM) of 2^{24} words of 16 bits each. Its input and output registers are 8 bits wide to accommodate the byte-serial data format. We used the two-phase bundled data convention of Figure 4 as the byte transfer protocol at the input and output of this memory controller. It contains seven parts: four event-controlled storage registers and three stages of logic between them.

Each stage of control in the memory controller operates much like one of the simple stages in micropipeline control of Figure 10. Like those of Figure 10, each stage includes a Muller C-element and each stage communicates only with adjacent stages using exactly the two-phase bundled data convention of Figure 4. The control for each stage is composed from the event logic elements shown in Figure 9: XORs, Muller C-elements, SELECTs, TOGGLEs, and CALLs. In some stages these elements are connected in loops to permit several actions to take place within the stage before it acknowledges data from a previous stage or requests service from a subsequent stage. Such loops pack and unpack data. A separate memory refresh procedure interrupts normal operation using an ARBITER.

The logic in each stage of the memory controller performs a different function. The first stage decodes byte-serial input from the 8-bit input register, converting it into a 54-bit parallel word containing all of the address, data, and control information required for a memory cycle. This stage accepts and acknowledges several bytes of input before requesting action from the next stage. Between the first and second logic stages is a 54-bit event-controlled register. The second stage uses each 54-bit parallel word to control one access to the external DRAM chips. When reading from memory, this access converts the 54-bit address and control information into a 16-bit data value. The control includes a timing model for the memory chips and waits for the memory cycle to finish before requesting action from the next stage. Between the second and third stage of logic is a 16-bit event-controlled register that captures the data output from the DRAM chips. The third stage repacks the 16-bit output data into byte-serial form and presents it at the output terminals through the 8-bit event-controlled output register.

This memory controller, operating as a pipeline, can be carrying out a memory access while concurrently packing up the previously accessed data and unpacking the byte-serial address and control information for the next access. Because the stages are free of a common clock and each runs at its own pace, the pipeline is elastic. The elasticity permits a memory cycle to occur whenever a single set of address and command values is presented at the input, which may require several input bytes, even if no further input is provided.

The behavior of micropipelines is a blend of combinatorial behavior and pipeline processing. Remember that event-controlled storage elements are transparent when empty, and can behave like combinatorial circuits, storing nothing. Thus when a micropipeline is empty it behaves just like a combinatorial circuit. After their data path delay, the decoders of Figures 18 and 19, if empty, faithfully report as output the correct one-out-of-N code for any given binary input. You can confirm this by examining Figures 18 and 19 and remembering that the switches are all drawn in the positions they occupy when the data path is empty. Notice that complete paths involving no storage are available from input to output. Similarly, the multiplier of Figure 20, if

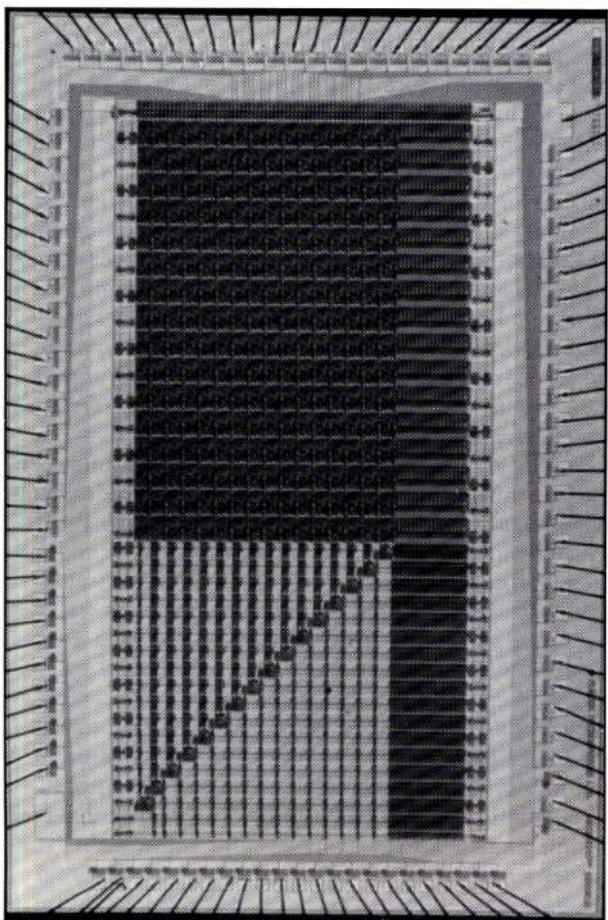


FIGURE 20. A Micropipeline Multiplier Chip

The experimental micropipeline multiplier shown in this photograph was built by Austek Microsystems. It multiplies pairs of 12-bit numbers using 24 stages of micropipeline; the first 12 stages are in the multiplication array, and the final 12 resolve the carry-save form of multiplier output. When empty, the multiplier acts just like a storage-free combinatorial multiplier, but when used as a pipeline it can accept up to 24 operand pairs before delivering its first product.

empty, faithfully reports as output the product of its input operands after its data path delay. This behavior makes the data path of a micropipeline easy to test.

The pipeline behavior of micropipelines is evident when they are given several inputs in rapid succession. Transition events on the request and acknowledge wires at the input end of the micropipeline serve to separate one input data element from another according to the two-phase bundled data convention of Figure 4. The "handshake" events on the request and acknowledge wires are like the rubber rods used in a grocery store check-out line to separate one customer's groceries from another's. Each request-acknowledge pair of events separates one data set from preceding or following data sets. The wave propagation properties of the Muller C-element control system move these data-separation events forward through the control circuits, and the control events force the data forward through

the event-controlled storage registers, just as motion of the conveyer belt in the grocery store moves rubber rods and groceries forward toward the cashier.

The pipeline behavior of micropipelines is also evident when they deliver several outputs in rapid succession. Again the "handshake" events on the request and acknowledge wires at the output end of the micropipeline serve to separate one output value from another. Again the two-phase bundled data convention of Figure 4 is used, each handshake bringing a new value to the output data terminals. Micropipelines can exhibit very high burst input and output data rates.

The two-phase bundled data convention of Figure 4 automatically takes care of the "full" and "empty" conditions of the micropipeline. If the micropipeline becomes full, it will delay the acknowledge event on its input end, thus preventing further input. Remember that the device feeding the micropipeline must conform to the two-phase bundled data convention of Figure 4, and therefore cannot change the input data until after it receives an acknowledgment for the present data. Similarly, if the micropipeline becomes empty, it will delay the request event at its output end, thus preventing the output device from taking erroneous data. At every internal stage of the micropipeline the same signalling convention applies. Thus if a section is full, it will automatically delay new data from earlier sections in the micropipeline.

OTHER DEVICES USING THE SAME PROTOCOL

The two-phase bundled data convention used in micropipelines can be applied to other types of devices as well. For example, one can build a ring-buffer FIFO whose interface characteristics are the same as those of the micropipeline FIFOs of Figure 15 or Figure 16. Such a device might use an external random access memory for the required storage and two address counters as pointers, one for reading and one for writing, to treat the memory as a ring buffer. It would compare the values of the two pointers to recognize if the ring buffer were full or empty, and if so to delay the handshake signals at its terminals. The ring buffer pointers and the full and empty signals that result from comparing them could be private internal signals not available outside the control.

If a single port memory is used in such a ring-buffer FIFO, an arbiter must be used to decide whether the next memory cycle will be devoted to reading or to writing. Arbitration is required because a single resource, namely the memory access port, must be shared between two independently timed processes, the input process and the output process. If, at precisely the same instant, the input process delivers a new input value and the output process asks for a new output value, the arbiter must decide cleanly which request to service first. A transition logic control for such a ring-buffer FIFO is shown in Figure 21.

Although the circuit of Figure 21 looks like a flow diagram, it is in fact a circuit. It is composed of simple transition control modules, all of which use transition

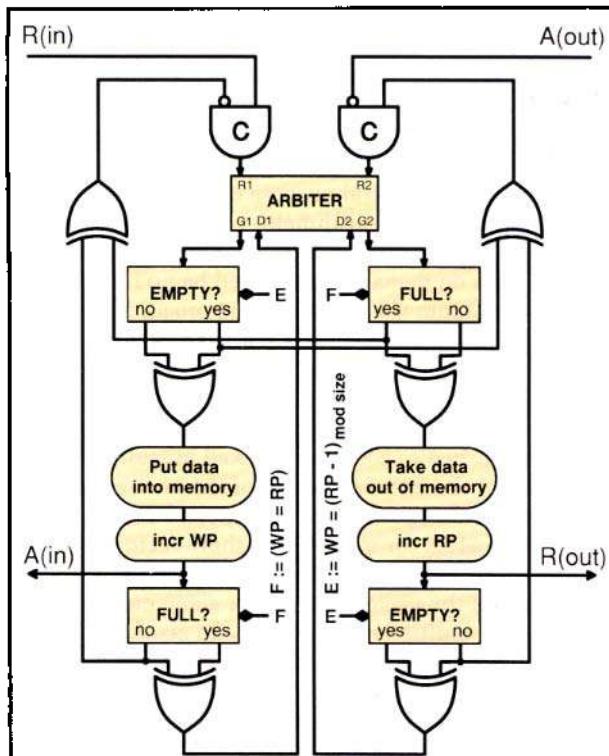


FIGURE 21. Ring-Buffer FIFO Control Logic

The control logic for a ring-buffer FIFO can be composed from the event logic modules shown in Figure 9. Except for the test values, all wires shown here carry event signals; the data path, the address pointers and the memory are not shown. In each of the four SELECT modules I have written the name of its test; the wires labeled "E" and "F" carry the required Boolean values. The functions described in the four lozenges include memory access and incrementing the read and write pointers, RP and WP. Although this figure looks like a block diagram, it is actually a circuit ready for direct implementation. It has been proven [5] that an external observer cannot distinguish this ring-buffer FIFO control circuit from the micropipeline control circuit of Figure 10.

signalling. Because these modules are insensitive to delay, composing them into circuits is much like drawing flow diagrams. Using tools developed by David Dill, my colleague, Bob Sproull, proved that if the FIFO controls of Figures 10 and 21 work at all, then for equivalent memory sizes they are functionally equivalent [5]. We can be assured, therefore, that such a ring-buffer FIFO and the micropipeline FIFO are interchangeable. The ability to make such proofs is one of the appealing things about the transition-signalling conceptual framework.

Using the two-phase bundled data convention of Figure 4 between micropipeline stages leaves wide latitude to make individual stages perform their functions in diverse ways. For example, a pipeline device for arithmetic normalization can be built with many stages or with a single stage. The multi-stage version performs a single bit shift in each stage, has very high throughput, exhibits long latency, and provides much buffer space.

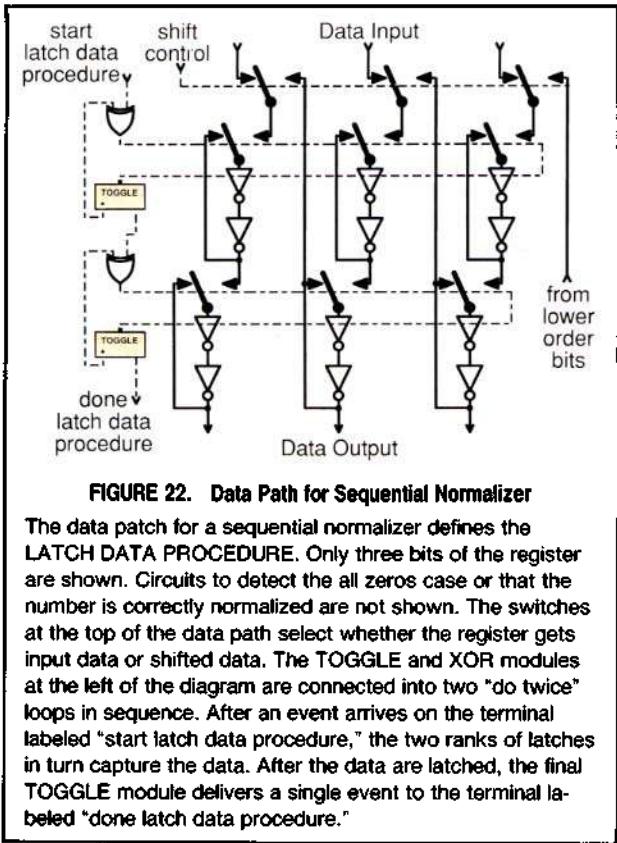
The single stage device performs its shifts sequentially, has reduced throughput and buffer space, but requires substantially less circuitry.

Three bits of a data path for such a single stage sequential normalizer are shown in Figure 22. Two registers of the form illustrated in Figure 14 are used in series to capture and hold the data. Switches at the top of the diagram select whether input data or shifted data enter the registers. The XOR and TOGGLE modules at the left of this circuit serve a similar role to those in Figure 14. Each event on the wire labeled "start latch data procedure" produces two events on each latch control wire and thus flips the register switches out of the position shown and then back into the position shown, capturing a new data value in the register. You should think of Figure 22 as the definition of the LATCH DATA PROCEDURE. This procedure has one input parameter, the "shift control" signal shown, and one output parameter, the "normalized or all zero" signal, which is generated by circuits omitted from the figure.

The sequential form of normalizer operates just as would a normalization program for a computer able to shift left only one place at a time. The control circuit is shown in Figure 23. At the top of the figure is a Muller C-element, similar to those we have seen in other micropipeline stages. After an event leaves the Muller C-element, its first action is to capture an input datum by using the upper client terminal, R1, of the CALL module to access the latch data procedure represented by the lozenge and defined in Figure 22. When the latch data procedure is done, it returns an event to the D terminal of the CALL module, which in turn returns an event to its D1 terminal. Thus shortly after the data are captured and before they are normalized, the control produces an event on its input acknowledge wire, A(in). From the point of view of a micropipeline stage, the rest of the algorithm below A(in) is just a delay before R(out).

After capturing the input datum, the control uses a while loop to shift the data into normalized form. The while loop contains an XOR module, a SELECT module, and the latch data procedure via the lower client terminals of the CALL module. An event circulates around the while loop and through the latch data procedure as long as the data are not yet normalized, causing one shift per trip around the loop. The time between shifts is established by the loop delay, and may be as fast or as slow as the circuits involved. When the while loop finishes, the event exits from the loop via the "true" output of the SELECT module. Thus when shifting is complete the control makes an event on the output request wire, R(out).

The shift control wire shown at the left of Figure 23 deserves special mention. We can think about it in two ways. First, thinking in terms of events, we should put an event on this control wire just before the while loop starts to flip the shift control switches into the shift position, and we need another event on it when the while loop finishes to flip the switches back into the



input position. The two inputs to the XOR element that drives the shift control wire serve to bracket the while loop and thus deliver the two required events. The other way of thinking about the shift control considers the value of the XOR module output. So long as the while loop is active, its input and output control terminals will be in different states, and thus the output of the XOR module will be high, setting the switches in the correct position for shifting.

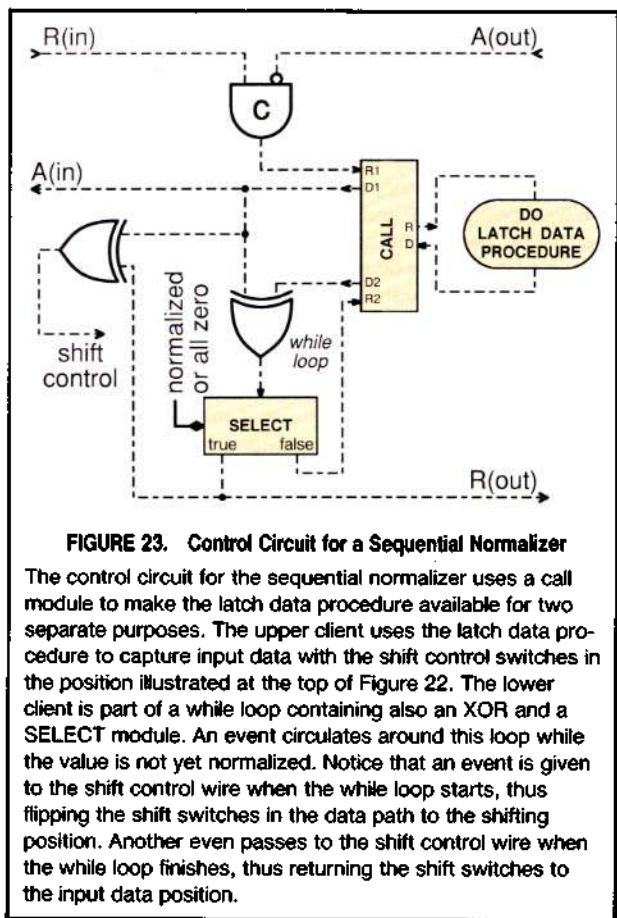
Designing control circuits like the ones illustrated here is rather like making block diagrams for programs. Not only do the event logic modules provide conditionals, procedure call, and other elements familiar to programmers, but also their response to events makes them easy to compose into loops and other structures similar to those found in programs. Using the form of the micropipeline control, it is also easy to build concurrent processing devices. We and others have built and tested libraries of such event logic modules and found them remarkably easy to use; the similarity of composing event-driven modules and programming has been recognized and used to advantage in a few places at least since the macromodule project [2, 3] during the 1960s. It provides, I think, an exciting alternative to conventional hardware design.

MICROPIPELINES IN GENERAL PURPOSE COMPUTING

General purpose computing machines use pipelines for two purposes: computation data paths and instruction

decoding. They could also use pipelines in memory fetch operations if common memory parts and controllers were built using the micropipeline framework. Let us consider each of these three applications in turn to see how the micropipeline framework might improve system performance or usability.

Let us imagine a general purpose computing machine with micropipelines for arithmetic vector processing. Because the micropipelines provide an amount of storage that varies on demand, there need be no fixed vector length built into the machine. The program would be free to load vectors of any length, up to a maximum, into such a micropipeline, and subsequently unload the results. Using a micropipeline adder, for example, a program might pile in a set of address and offset addition tasks required to compute indexed memory references and use the internal storage of the micropipeline to hold the resulting sequence of addresses until needed. A program for multiplying short vectors by small matrices, an operation useful in computer graphics, might load the vector and matrix elements into a micropipeline multiplier followed by an accumulator. Vectors of 2, 3, or 4 components could be handled easily and efficiently by the same equipment. Moreover, because input and output operations might be separated in time, the indexing required for memory access might be simplified.



Perhaps the most important applications of micro-pipelines will involve operations in which the vector length changes. One such example is the clipping operation widely used in computer graphics [14, 15]. The clipping operation removes the parts of a set of objects that lie outside a reference window. Clipping may result in an increase or decrease in the number of objects in the set. Because whole objects may be removed, there may be less output than input, but because connected edges may also be broken into multiple pieces, there may also be more output than input. Such a clipping device with very simple interface characteristics can be built using the micropipeline framework.

Sorting is another important application for micro-pipelines in which the vector length changes. Micro-pipelines can be applied to both the partitioning and merging operations used in sorting. For partitioning, suppose that two micropipelines are connected to the outputs of a rapid micropipeline partitioner. Given a vector of input values, the partitioner can separate them into two output vectors according to some partitioning criterion, delivering elements from each vector into the corresponding output micropipeline. Of course, the number of elements in each of the two output vectors is data dependent, but the micropipelines are elastic and can easily accommodate variable length vectors by increasing or decreasing on demand the amount of storage available. The outputs of the two micropipelines can deliver the partitioned values without any need for priming or flushing.

Micropipelines can be applied to the merging operation as well. In this case two micropipelines for storing input vectors are connected to the two inputs of a merging device. This merging device can make whatever comparison is appropriate between the data values it is presented and select one of them for output. The elastic property of the input micropipelines permits the merging device to take data from either of them in whatever sequence the data values require. Partitioning and merging devices can be useful in signal-processing pipelines as well, for example, to divide a workload between several parallel pipelines.

Let us now turn from arithmetic to instruction processing. Pipeline instruction processing has become very common, and with reduced instruction set computer (RISC) architectures, is by now very well understood. One of its side effects is called "delayed branch." This name describes the fact that some precise number of instructions, for example exactly 2, will be performed in sequence after each jump or conditional jump instruction before the branch actually takes effect. These "overhang" instructions are necessary to keep the inelastic instruction processing pipeline busy while the new jump address takes effect. If nothing useful can be done in these overhang instructions, NOPs must be inserted as input to the instruction processing pipeline while it completes work on the jump instruction.

Let us imagine a micropipeline instruction processor. Such a processor can avoid the requirement for over-

hang instructions, but permit them to be included for additional speed. Although the micropipeline latency may create a time delay equivalent to two instructions after a jump, such a processor need not impose the storage cost of NOPs. If there is nothing useful to do, the NOPs can be omitted to save the storage. If some other number of instructions can usefully be done after the jump, for example, one or three, they may be inserted. By expanding or contracting the amount of storage used in the instruction processing pipeline on demand, the micropipeline framework can increase the programmer's flexibility. No longer does the pipeline have to contain exactly a fixed number of storage cells.

Condition codes can usefully be passed through a micropipeline. Conditions such as arithmetic comparison or parity for which a pipeline offers high throughput can be computed in vector fashion. For maximum throughput, the program should insert other operations between computing a condition and testing its result. If there is nothing useful to do between computing a condition and testing its result, intervening instructions may be omitted and the condition micropipeline behaves like a combinatorial circuit. Such a program may suffer the delay of the micropipeline latency, but will work properly.

With a micropipeline for storing condition codes, a program can compute several conditions before testing the first of them. The condition codes remain in the micropipeline in first-in-first-out sequence until tested. This is particularly useful in multi-way decoding trees, for example where three conditions control an 8-way branch. Instructions to compute each condition are required only once, and the three codes thus generated are stored in the micropipeline until tested in the branch tree. In conventional machines the instructions that compute the second and third conditions must be duplicated in each branch of the test tree.

Finally, memory systems obviously fit well into the micropipeline framework. One might design a dynamic random access memory (DRAM) part using a micro-pipeline. Such a memory part can provide at least a factor of two improvement in throughput over conventional DRAM parts. This improvement comes about because such a memory part can access its memory array concurrently with decoding the next address and with driving its data output pin or pins with the previously retrieved data. Such an improved part requires relatively little additional circuitry, since many of the actions in a DRAM are already driven from an internal timing chain. Only suitable event-controlled latches and Muller C-elements to form a micropipeline need be added to the existing DRAM logic, control mechanisms, and delay models. When concurrency is not needed, the micropipeline will be empty, making the event-controlled storage elements transparent, and permitting the micropipeline DRAM part to behave much like the one-cycle-at-a-time DRAM parts now in widespread use.

One might worry that a DRAM part with a micro-pipeline would require four control wires, two at the

input port and two at the output port, where existing parts have only two, called RAS and CAS. This is not so, because it will prove better to use an external timing model of the DRAM behavior, based on the manufacturer's worst case specifications, rather than to have each and every DRAM part in a system report completion on its own. The pins for completion signals at both input and output port can be omitted from the individual DRAM parts, because the external timing model provides the two missing completion signals on behalf of the entire memory system, using its model of the DRAM behavior to provide suitable delays. The input port of individual DRAM parts needs only the request wire, and the output port of individual DRAM parts needs only the acknowledge wire. Events on these two control wires respectively tell the DRAM part when to accept new address information or data to be written, and when to present a new output value. The external timing model will itself be a micropipeline built with stage delays that equal or exceed, stage by stage, the corresponding delays in the micropipeline in the DRAM parts.

Cache memories also fit well into the micropipeline framework. A very high throughput cache built within this framework can perform decode, detect "hits," and drive its output all concurrently. Such a cache memory has two interface pairs, both using a signalling convention similar to the two-phase bundled data convention of Figure 4. One pair of interfaces connects the cache to the processor and the other pair connects it to memory, as shown in Figure 24. The pair of interfaces between the processor and the cache should be identical to the pair of interfaces between the cache and the memory, so that the system can operate with or without the cache as shown in Figures 24 and 25.

The processor, cache, and memory, taken together form a micropipeline. Memory requests from the processor flow into the cache and back in micropipeline fashion, going to memory and back only when necessary. The processor can give the cache several memory requests concurrently before getting any data back. For highest throughput, the processor should deliver a continuous stream of memory requests, but it operates correctly, albeit at reduced throughput, if it gives only one request at a time. Because the two-phase bundled data convention of Figure 4 permits either sender or receiver to delay the next transaction arbitrarily, cache or memory access delays automatically delay subsequent requests from the processor. Similarly, the part of the processor that consumes memory data waits however long is required for the events that signal the presence of valid data.

If the cache does not contain the required information, it passes the request on to the memory. In this case the processor suffers the additional delay required to fetch information from memory. Because the processor accepts data from the cache only when it detects a validating request event, the processor easily accommodates to any additional memory delay. In fact, if the

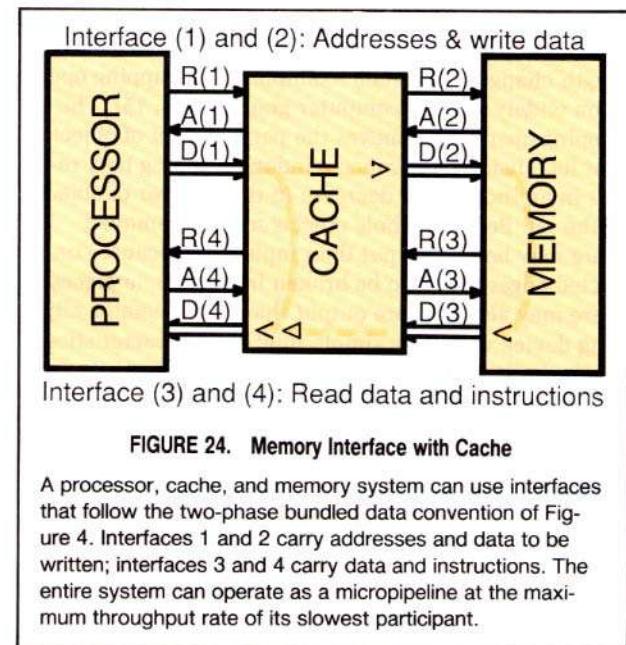


FIGURE 24. Memory Interface with Cache

A processor, cache, and memory system can use interfaces that follow the two-phase bundled data convention of Figure 4. Interfaces 1 and 2 carry addresses and data to be written; interfaces 3 and 4 carry data and instructions. The entire system can operate as a micropipeline at the maximum throughput rate of its slowest participant.

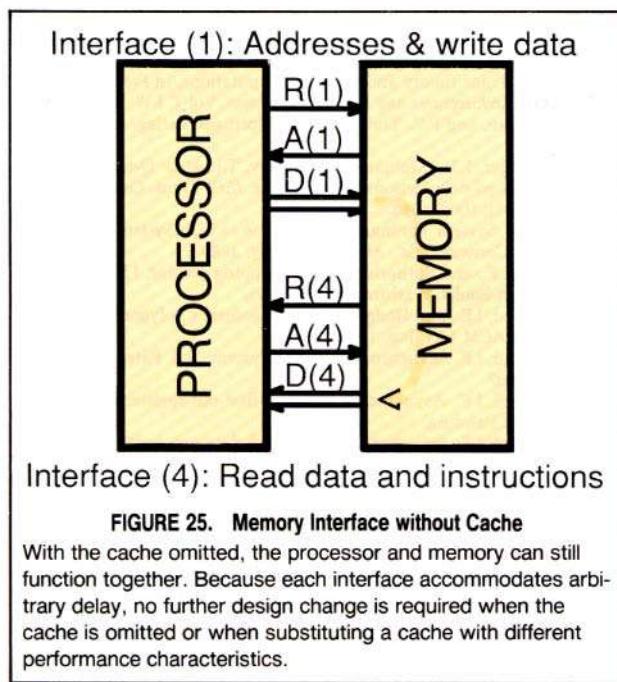
cache sent every request to the memory, or if the cache were omitted, the system would still operate properly, albeit at reduced throughput.

This leads me to the most important implication of micropipelines. Because they use event-controlled interfaces rather than a common clock, micropipelines with different inherent speeds can be composed directly into systems that function correctly, albeit at the speed of the slowest part. If the cache of Figure 24 were omitted or replaced with a cache with different cost and performance characteristics, the system would still operate correctly. Similarly the processor performance or the memory performance can be upgraded and the system will still work, taking advantage of any available speed improvements.

CONCLUSION

FIFOs and pipelines are simple to design and easy to understand in the transition-signalling conceptual framework. They are relatively difficult to design within the clocked-logic conceptual framework. By abandoning clocked logic in favor of transition signalling, one is able to make very simple micropipelines that assemble easily into larger structures. The change in conceptual framework suggested here simplifies system design because simple modules and compositions of them can be further composed into large systems.

The composability offered by micropipelines and transition signalling may be their most important property. Complex functions are easy to compose from simple modules that provide basic functions already familiar in programming. More complex systems can be built by composing them as a hierarchy of the basic modules and previously designed compositions. Even if the basic building blocks were hard to design, and they no longer are, they would be worthwhile, because they are so



easy to compose into systems.

This same composability offers a simple way to upgrade system performance as improved circuitry becomes available. Event-driven interface protocols permit old components to be replaced by new ones with improved throughput, latency, or cost characteristics. Because the handshake used here automatically takes care of delays in delivering or making use of data, such replacements can be made with assurance that the system will still operate properly. On the other hand, large systems built in the clocked-logic conceptual framework resist incremental improvement, because any increase in clock speed must be accommodated throughout the system. As improvements are made to systems built as I have outlined here, one can expect that the slowest or most expensive parts of a system will be replaced first, and thus that each replacement will improve system performance or decrease system cost. Thus the transition-signalling conceptual framework, micropipelines, and the two-phase bundled data convention of Figure 4 taken together not only simplify initial system design but also permit rapid mid-life upgrade of systems as new technology becomes available.

I hope that this lecture may help system design to keep pace with advancing component technology. Today, new integrated circuit technology makes available significant improvements in cost or performance every six months or so. It is often difficult to make use of such improved performance, because speeding up the clock in an entire system is a formidable task fraught with dangers. Today's system designers, constrained by the clocked-logic conceptual framework, take several years to produce a new system. Thus the systems being sold may lag by several years the potential speed or cost benefits offered by the most modern technology. I be-

lieve that the micropipeline framework that I have described here can reduce the opportunity cost imposed by the clocked-logic conceptual framework.

Acknowledgments. The transition-signalling conceptual framework has been used in a few places over a long period of time. I know of early work at the University of Illinois by David Muller, at the University of Utah by Al Davis, and at the Massachusetts Institute of Technology by Jack Dennis; I apologize in advance for omitting mention of other projects.

I owe my own education in the transition-signalling conceptual framework to a few able people. I first became aware of transition signalling in the early 1960s when a group at Washington University in Saint Louis, led by Wes Clark, used it in the design of a set of macromodules [2,3]. I learned much more about it from Charles Seitz, now on the faculty at Caltech, over a dozen years starting in 1966 when, as an MIT graduate student, he taught me most of what I know about digital design. We worked together at Harvard and at the Evans and Sutherland Computer Corporation using an almost-correct version of micropipelines in processors for computer graphics, including the original "clipping divider" [14]. His chapter in the well-known Mead and Conway book on VLSI is one of the best presented and most accessible references on transition signalling [13].

Two other people have been important to my education. Most important to me over a long period of time is Bob Sproull, from whom I first started a lifelong education twenty-five years ago and of whose knowledge and ability I remain in awe. He has regularly fixed my thinking when it was fuzzy, and has made this lecture more accurate than I alone could have. During the past five years he and I led an "Asynchronous Systems Study" to learn and teach transition signalling. As part of it we designed micropipelines for various purposes, including those I have described here. We have taught our subject to a few hundred people, and we have two books in preparation. Finally, I want to mention Charles Molnar, whose group at Washington University continues the pioneering work I mentioned before. He has given unstintingly to my education not only his time and ideas but also his enthusiasm. His contributions to the transition-signalling conceptual framework include not only the absolutely essential synthesis method for logic modules [9] without which the new framework was difficult to use, but also many important parts of an overall mathematical theory, and new conceptions of useful circuits [12]. The theoretical work [6,10,11,19], in which he collaborates with Martin Rem's group in Eindhoven, is beginning to prove theorems about the correctness of systems designed in the transition-signalling conceptual framework.

The work reported here led to a broader "Asynchronous Systems Study," conducted by Sutherland, Sproull and Associates, Inc., and supported by six industrial sponsors: Apple Computer, Austek Microsystems Ltd., Digital Equipment Corp., Evans and Sutherland Com-

puter Corp., Floating Point Systems, and the Schlumberger Research Laboratory. We did the work with the cooperation of Carnegie Mellon University and Imperial College of the University of London. Erik Brunvand, Ed Frank, Ian Jones, Charles Molnar, and Bert Sutherland collaborated with us. We were able to test micropipeline circuits fabricated for us by the MOSIS integrated circuit fabrication service operated by the Information Sciences Institute of the University of Southern California. We are proud to have been the very first commercial MOSIS client.

REFERENCES

1. Chaney, T.J., and Molnar, C.E. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput.* C-22, 4 (Apr. 1973), 421-422.
2. Clark, W.A. Macromodular computer systems. In *Proceedings of the Spring Joint Computer Conference*, AFIPS, April 1967.
3. Clark, W.A., and Molnar, C.E. Macromodular computer systems. *Computers in Biomedical Research*, Vol. 4. R. Stacy and B. Waxman, Eds., Academic Press, New York, 1974, 45-85.
4. Dally, W.J., Seitz, C.L. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.* 36, 5 (May 1987), 547-553.
5. Dill, D.L., Nowick, S.M., and Sproull, R.F. Specification and automatic verification of self-timed queues. Computer Systems Laboratory Report, Stanford University, 1988.
6. Ebergen, J.C. Translating programs into delay-insensitive circuits. Ph.D. dissertation, Eindhoven University of Technology, 1987.
7. Levy, J.V. Buses, the skeleton of computer structures. In *Computer Engineering*, C.G. Bell, J.C. Mudge, and J.E. McNamara, Eds., Digital Press, 1978.
8. Miller, R.E. "Sequential Circuits", Chapter 10, In *Switching Theory*, Vol 2, Wiley, NY, 1965.
9. Molnar, C.E., Fang, T.P., and Rosenberger, F.U. Synthesis of delay-insensitive modules. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, H. Fuchs, Ed., Computer Science Press, 1985.
10. Rem, M., van de Snepscheut, J.L.A., and Udding, J.T. Trace theory and the definition of hierarchical components. In *Proceedings of the Caltech Conference on VLSI*, 1983.
11. Rem, M. Trace theory and systolic computations. In *Proc. PARLE (Parallel Architectures and Languages Europe)*, Vol 1, J.W. deBakker, A.I. Nijman, and P.C. Treleaven, Eds., Springer-Verlag, 1987, pp. 14-34.
12. Rosenberger, F.U., Molnar, C.E., Chaney, T.J., et al. Q-modules: Locally clocked delay-insensitive modules. *IEEE Trans. Comput.* 37, 9 (Sept. 1988), 1005-1018.
13. Seitz, C.L. System Timing. In *Introduction to VLSI Systems*. C.A. Mead and L.A. Conway, Eds., Addison-Wesley, 1980.
14. Sproull, R.F., and Sutherland, I.E. A clipping divider. *FJCC* 1968, Thompson Books, Washington, D.C., 765.
15. Sutherland, I.E., and Hodgman, G.W. Reentrant polygon clipping. *Commun. ACM* 17, 1 (Jan. 1974), 32-42.
16. Sutherland, I.E. Asynchronous queue system. U.S. Patent 4,679,213, July 7, 1987.
17. Sutherland, I.E. Asynchronous first-in-first-out register structure. US Patent Pending.
18. Sutherland, I.E. Asynchronous pipelined data processing system. US Patent pending.
19. Udding, J.T. A formal model for defining assifying delay-insensitive circuits and systems. *J. Distrib. Comput.* 1, 1986, 197-204.

CR Categories and Subject Descriptors: B.2.1 [Arithmetic and Logic Structures]: Design Styles—pipeline; B.5.1 [Register Transfer-Level Implementation]: Design—styles[e.g., parallel, pipeline, special-purpose]; B.7.1 [Integrated Circuits]: Types and Design Styles—Input/Output circuits

General Terms: Design

Additional Key Words and Phrases: Asynchronous handshake, FIFO, pipeline processing, transition signalling

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Offers you exciting, significant, and original work in all aspects of the use and development of computer graphics...

Makes liberal use of high-quality color in many articles

acm Transactions on Graphics

Editor-in-Chief John C. Beatty
University of Waterloo, Ontario, Canada

Keep pace with the fast-breaking advances in computer graphics with *ACM Transactions on Graphics* (TOG). Offering exciting, significant, and original work in all aspects of the use and development of computer graphics, TOG covers topics such as image synthesis, geometric modeling, CAD/CAM/CAE, algorithm design and analysis, graphics programming language design and packages, person-machine interaction techniques, computer graphics hardware, and design and implementation of applications systems.

Making liberal use of high-quality color images in many articles, TOG is divided into two sections: Research Contributions and Practice and Experience. A unique feature, "The Interactive Technique Notebook," thumbnails such techniques and serves as a source for designers of interactive graphic applications programs.

Whether you are just discovering the diverse possibilities in the field or are already an expert, TOG is the journal for you. Published quarterly.

ISSN: 0730-0301

Included in *Science Abstracts*, *Automatic Subject Citation Alert*, *Computer Literature Index*, *Computing Reviews*, *Compumath Citation Index*, *Computer Aided Design/Computer Aided Manufacturing*, *Ergonomics Abstracts* and *International Aerospace Abstracts*.

Order No. 109000

Subscriptions: \$75.00/year — Mbrs. \$26.00

Single Issues: \$27.00 — Mbrs. \$14.00

Back Volumes: \$108.00 — Mbrs. \$56.00

Student Mbrs. \$21/year

Please send all orders and inquiries to:
P.O. Box 12115
Church Street Station
New York, NY 10249

