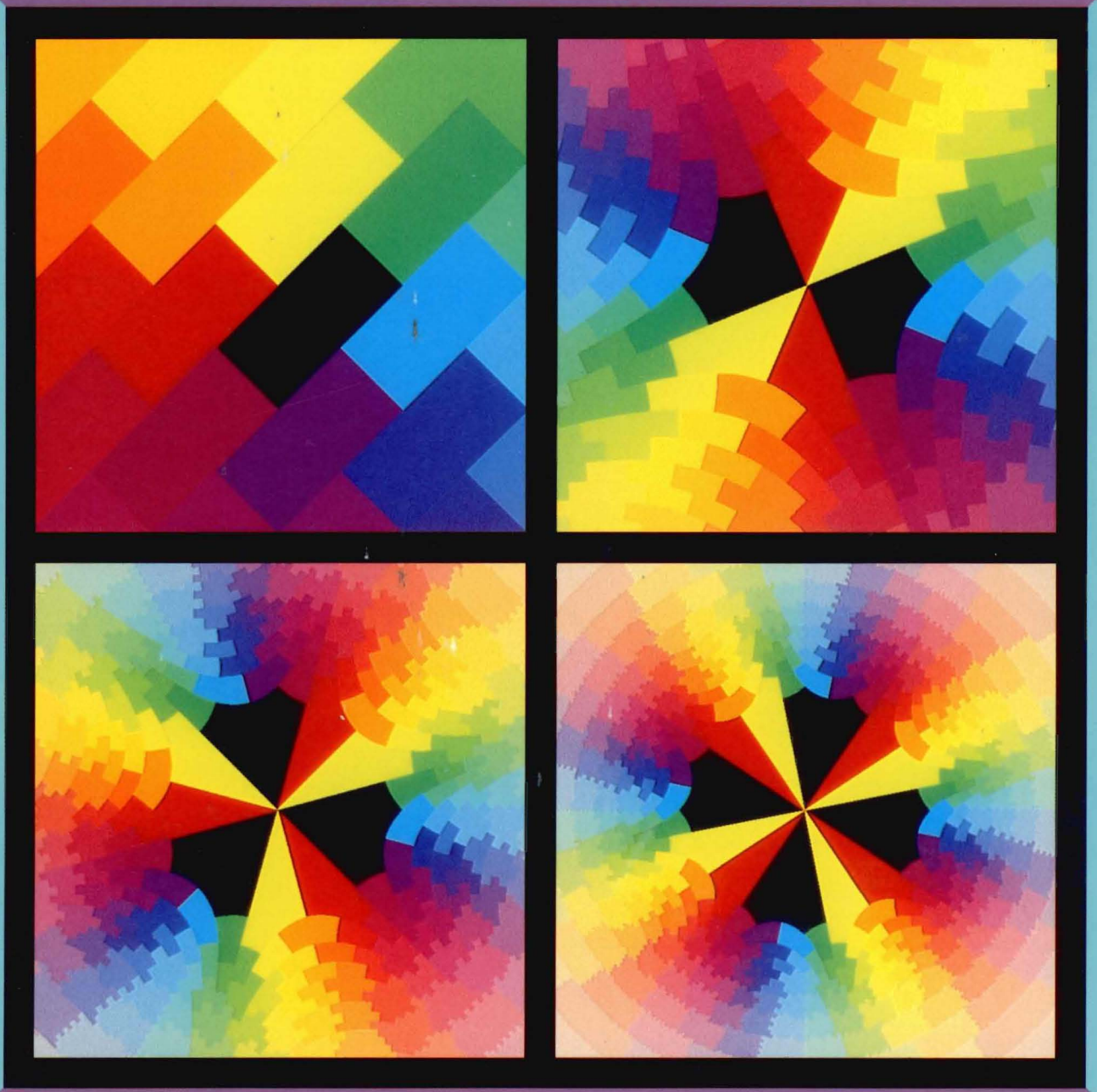
Twenty-fifth Anniversary

APL

IBM

Systems Journal

Vol. 30, No. 4, 1991



The cover and this special issue celebrate the 25th anniversary of APL. The graphics shown on the cover are mappings of the complex floor function for the first four integral powers of the complex plane. The mappings illustrate how APL is used as a "tool for thought" to analyze and visualize intricate problems through its interactive support for experimentation. They depict phase, magnitude, poles, zeros, and branch cuts. The phase is shown by hue in the color wheel, with cyan being positive real. The magnitude is shown by color saturation and brightness: magnitudes less than one get progressively darker as saturation increases, and eventually become black; magnitudes greater than one get progressively lighter as saturation increases, and eventually become white.

The cover art was created by David A. Rabenhorst at the IBM Research Division in Hawthorne, New York. He used the native support for complex numbers and the power of image processing in APL2 and AIX on the IBM RISC System/6000.

The *IBM Systems Journal* is a refereed technical journal published quarterly by International Business Machines Corporation, Armonk, New York 10504 U.S.A. Officers: John F. Akers, Chairman of the Board; Jack D. Kuehler, President; Robert M. Ripp, Treasurer; John E. Hickey, Secretary.

The *Journal* welcomes submissions and subscriptions from members of the worldwide professional and academic community who are interested in advances in software and systems. A guide for authors was published in Volume 29, Number 4 (1990) and is available as a reprint by order number G321-5419 (see below). Please send manuscripts and letters to the Editor, *IBM Systems Journal*, Armonk, New York 10504 U.S.A.

Subscription rate: $20.00 per calendar year (single copies $6.00). To order subscriptions or report change of address, write to IBM, P.O. Box 3033, Southeastern, Pennsylvania 19398 U.S.A. Reprints of articles in this issue at $1.00 each may be ordered from IBM branch offices using the reprint order number on the last page of each article. Complete copies of this and other issues may also be ordered from branch offices using the order number on the back cover.

Subscriptions may be entered or cancelled by IBM employees in the U.S.A. via VM/CMS by using the command JOURNALS, or by mailing the Journals Subscription Card (ZM08-2203). Change of address for IBM employees is handled automatically by the IBM Corporate Employee Resource Information System and need not be reported to the *Journal*. IBM employees outside the U.S.A. may subscribe through their Country Literature Coordinators or local IBM libraries.

TryAPL2

TryAPL2 is a free version of IBM APL2/PC for demonstration and education. It is suitable for classroom use. No commercial use is permitted.

Return this card before June 1, 1992 to receive your free copy of TryAPL2 on a 3.5" 720K disk.

Name: _____ Phone: _____

Title: _____

Company: _____

Address: _____

_____

City: _____ State: _____

Country: _____ ZIP: _____

International Business Machines Corporation
Direct Response Marketing
Department 787
P. O. Box 3974
Peoria, IL 61612-8849

TryAPL2

TryAPL2 is a free version of IBM APL2/PC for demonstration and education. It is suitable for classroom use. No commercial use is permitted.

TryAPL2 contains sample workspaces for graphics, statistics, calendar computations, operations research, and artificial intelligence.

To receive your free copy of TryAPL2 on a 3.5" 720K disk, send your complete mailing address by June 1, 1992 to:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1016 PEORIA, IL

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Direct Response Marketing
Department 787
P. O. Box 3974
Peoria, IL 61612-8849

IBM
Systems Journal
Vol. 30, No. 4, 1991

 ISSN 18-8670. Printed in U.S.A.

Preface

The first APL workspace became available at IBM on November 27, 1966, making 1991 the twenty-fifth anniversary of APL. The *IBM Systems Journal* joins in the anniversary celebration by presenting 12 papers and one essay covering APL's history, implementation, and applications. We are indebted to R. P. Polivka of the IBM Data Systems Division in Poughkeepsie, New York, for his extensive contributions to the planning and development of this issue, including the solicitation of numerous papers and suggestions for referees. We also commend and thank J. McGrew of the IBM Application Solutions Division in Kingston, New York, for his considerable efforts in ensuring the proper appearance of APL throughout the issue.

The term APL is attributed to a suggestion made by A. Falkoff when a name was needed for the programming language that was to be built from the ideas in K. E. Iverson's 1962 book entitled *A Programming Language.* Today, after many generations of the language and implementations, APL2 is IBM's strategic interactive programming language, serving IBM and its customers in a wide range of applications and across a broad spectrum of implementations.

The papers and essay begin with a history and an introduction to APL, then progress through a set of papers on APL systems and a set on applications, ending with an exploration of the importance of symbols and a look forward from Iverson's unique vantage point.

The first paper, by Falkoff, traces the genealogy of the IBM family of APL systems. His perspective stems from his place as one of the first, foremost, and current advocates of APL. He describes the interplay between language constructs, implementation methods, and evolution for the breadth of IBM APL systems.

Brown and Crowder introduce the essential features of APL2, IBM's current APL offering. The authors show, through examples, the use of arrays and functions, and show how the arrays (APL's data structures) control the flow of execution of a program.

Programming languages exist in close association with the language environment designed for their use. In the first paper in the set on APL systems, Wheatley discusses the issue of connectivity among APL2, its environment, and other programming languages. From the point of view of APL2, there are three major facilities that permit communication beyond the APL workspace: system variables and functions, shared variables, and name association. Each is presented, along with the historical setting.

Most APL systems have depended on the storage management technique known as garbage collection. This strategy has become less effective as virtual and real storage have grown dramatically. APL2 Version 2 takes a new approach: a quickcell scheme for small data items, a variation of the buddy system, and a bit map scheme for large blocks of storage. Trimble shows how this provides a better means of storage management.

Jensen and Beaty present the results and the experience of building an X Window System** interface for APL2 (called APL2/X). They also present a C interface for all IBM APL2 systems (called APL2-to-C), which was created in order to support the APL2/X effort. Following an overview of the X Window System and the interface design criteria, the authors detail the APL2/X and APL2-to-C interfaces, concluding with a sample program.

The APL IL Interpreter Generator has contributed to the successful and rapid proliferation of APL systems. Alfonseca, Selby, and Wilks describe IL and the use of it to generate new APL interpreters. To date IL has been used to create nine IBM products, with as little as 13 person-weeks of effort.

APL, with its array orientation, would appear to be a natural candidate for use in parallel expression

and computing. Willhoft analyzes each APL construct for its potential parallelism. Through argument and studies of examples, he shows that APL has a high degree of parallelism, both in its constructs and in its common uses. The types of parallelism examined are data, algorithm, data flow, and task. Suggestions are made for improving the language and implementations to further increase parallelism.

Turning to papers on APL applications, Jordan and Friis describe the application of APL2 to music, both for building music software and as a musical notation. Examples are given that show how frequency, pitch, tempo, loudness, chords, and passages can be represented in APL. The authors claim that the iconic nature of APL2 is well suited to musical expression.

Verifying that an implementation of a new architecture indeed matches its functional specification usually involves the use of test generators. The IBM RISC System/6000* was tested in that way by the random test program generator (RTPG), built in APL for that purpose. Aharon, Bar-David, Dorfman, Gofman, Leibowitz, and Schwartzburd present the concepts and implementation of RTPG. They discuss the advantages of using an interactive language in test situations, where many changes are made with a need for rapid test creation, and the suitability of using APL to represent computer architectures.

Thomson describes the efforts of a group of academic and industrial statisticians in the United Kingdom, with the support of the British APL Association, to build on the popularity of APL for statistics and on its ability to express specifications of mathematical functions. They are creating the APL Statistics Library (ASL), which will contain standardized APL specifications of statistical functions. The author describes the philosophy of ASL code and documentation and illustrates how it provides a medium for algorithmic discussion among statisticians. The paper concludes with a demonstration of how advanced functions can be readily and reliably built using standardized ones from ASL.

Alfonseca summarizes his work on the application of APL to the fields of logic programming and artificial intelligence, neural networks, and object-oriented programming and hypertext. The paper argues that APL is applicable to a broad range of modern programming challenges.

Leaving the papers on APL applications, McIntyre describes APL from the perspective of symbolic languages throughout history, including our number system, many ancient written languages, and much of mathematics. He finds that APL, a symbolic language, is an "intellectual triumph." This paper grew out of an invited talk at APL83 in Washington, D.C., where the author presented a detailed history of the evolution of symbols.

The issue closes with an essay by Iverson that traces the development of his rationale for the APL notation, beginning with his original motive: creation of a tool for writing and teaching about data processing. Much of the essay is devoted to a discussion of the J language, which has evolved from his earlier work with APL. Iverson continues to pursue language styles and constructs that would be accessible to wide audiences.

There have been two changes to the form of the *Journal*. The first is the use of asterisks to signify a trademark or registered trademark. The appropriate designation for each term is shown just before the list of references in each paper. The second is the inclusion of the date on which the paper was accepted for publication by the editors (following editorial and peer review, and author revisions) and after which content would not have been materially changed. That date is shown just after the list of references in each paper.

The next issue of the *Journal* will contain several papers on the Optimization Subroutine Library (OSL) and others on such subjects as a portable model for the design of device drivers in OS/2*.

Gene F. Hoffnagle
Editor

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Massachusetts Institute of Technology.

The IBM family of APL systems

by A. D. Falkoff

The developmental history of IBM subfamilies of APL systems is traced in this paper, focusing on the inter-relationships among them and the methods of implementation used by the various groups involved. The language itself, and the way its evolution was managed, are also considered as factors influencing the development process. A chart is included that illustrates the evolution of mainframe and small machine programming products supporting APL, beginning in 1964 up to the present time.

In the 25 years since the first viable APL system was introduced outside of IBM, offerings of APL systems spanning most of the significant hardware families have been produced at a rate of more than one per year. These systems have been produced by small groups of designers and developers; at no time have there been more than about 20 people, company-wide, working on APL implementations at the same time. It is worth asking how this high productivity came about: the methods of implementation, the language itself, and the management of its evolution must have all been factors. In this paper, each of these factors is discussed as the history of the various subfamilies of APL systems is traced.

Figure 1[1] is provided to visually aid the reader in following this history. In this chart, shown later, the entries shaded in blue are systems that achieved some form of product status; the others are developmental or experimental systems, which in many cases had significant IBM internal usage. The vertical coordinate is a time line, starting with 1964 at the top. On the horizontal axis there are six columns. In general, each column is devoted to a major subfamily of APL systems, or to the work of a particular implementation group. The fourth column does not fit this description; it shows work performed by different groups on two different subfamilies of systems, but they are connected in an interesting way that is described later. The directed lines on the chart indicate significant design influences or transport of code. Of course, they do not tell the whole story, as the actual transactions were usually more complex than can be so simply diagrammed.

Mainframe systems

The earliest work on APL and its forerunners, PAT and another called IVSYS, was done in IBM's Research Division. As has been reported elsewhere,[2] PAT (for Personalized Array Translator) was an interactive interpretive system using a limited set of array operations, coded for the IBM 1620 processor. It made clear that such a system could successfully be built, and it helped to motivate the design of the APL type element for the IBM Selectric* typewriter mechanism. IVSYS (for Iverson system) was the first attempt at a mainframe system.[3] It was an interpreter written in FORTRAN to run in batch mode on the IBM 7090 series of machines, and was rendered interactive by running it under an experimental time-sharing monitor[4] (TSM) on an IBM 7093 processor.

APL\360. No sooner did the original APL group have IVSYS running in late 1965, but they were told that the TSM project, which was not under their control, would be dismantled. If they were to continue experimenting with Iverson's ideas,[5] the only recourse was to undertake the development of a time-sharing system of their own, along with an interpreter, for the recently announced IBM System/360* line of machines. This work went remarkably well, resulting in an integrated system, APL\360,[6] with excellent performance characteristics.[7] The system was operational about three months after work was started, and the three implementers who did the bulk of the programming were later to receive an industry award for their work.[8] It is worth looking at the factors that contributed to this success.

First, although this was a new system, there were some important design decisions regarding the language, as well as some coding experience, carried over from the IVSYS project. Second, the design and development group was small and enthusiastic. This attracted help, both in the form of direct contributions to the coding and thoughtful feedback from early users. Third, the group did not try to do it all themselves. Mathematical functions were borrowed from the FORTRAN IV subroutine library, and ideas from other sources were adopted if considered useful. Fourth, the systematic nature of the language lent itself to a clean internal design of the interpreter. Fifth, the system was designed to be independent of the host operating system. The handling of input and output, management of user storage, and time-sharing functions were all built into the supervisor, which was tailored to the specific needs of the language processor, thus avoiding some of the complexities of more general systems. And last, even at that early stage, APL itself was used as a design tool. The supervisor, for example, was modeled in APL, and as the interpreter code progressed, the model was run on it for validation.

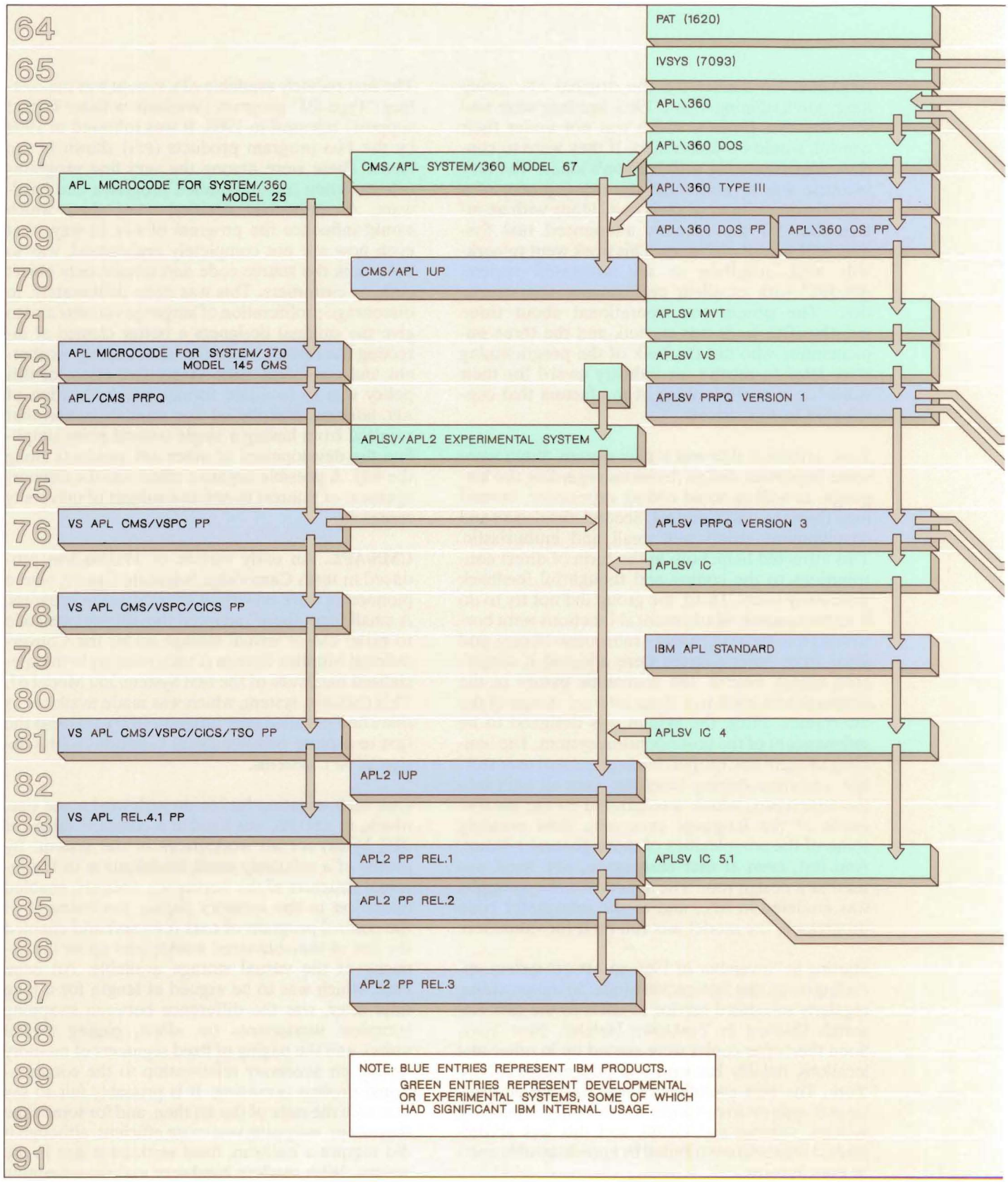
Starting in November of 1966 an APL\360 system operating on an IBM System/360 Model 50 was providing regularly scheduled service to users in the IBM Research Division in Yorktown Heights, New York. Soon thereafter copies were started up in other IBM locations, notably Endicott and Poughkeepsie, New York. The next evolutionary step was the development of systems to run under the two extant operating systems, DOS/360 and OS/360, and this was accomplished with help contributed by knowledgeable users in Poughkeepsie.

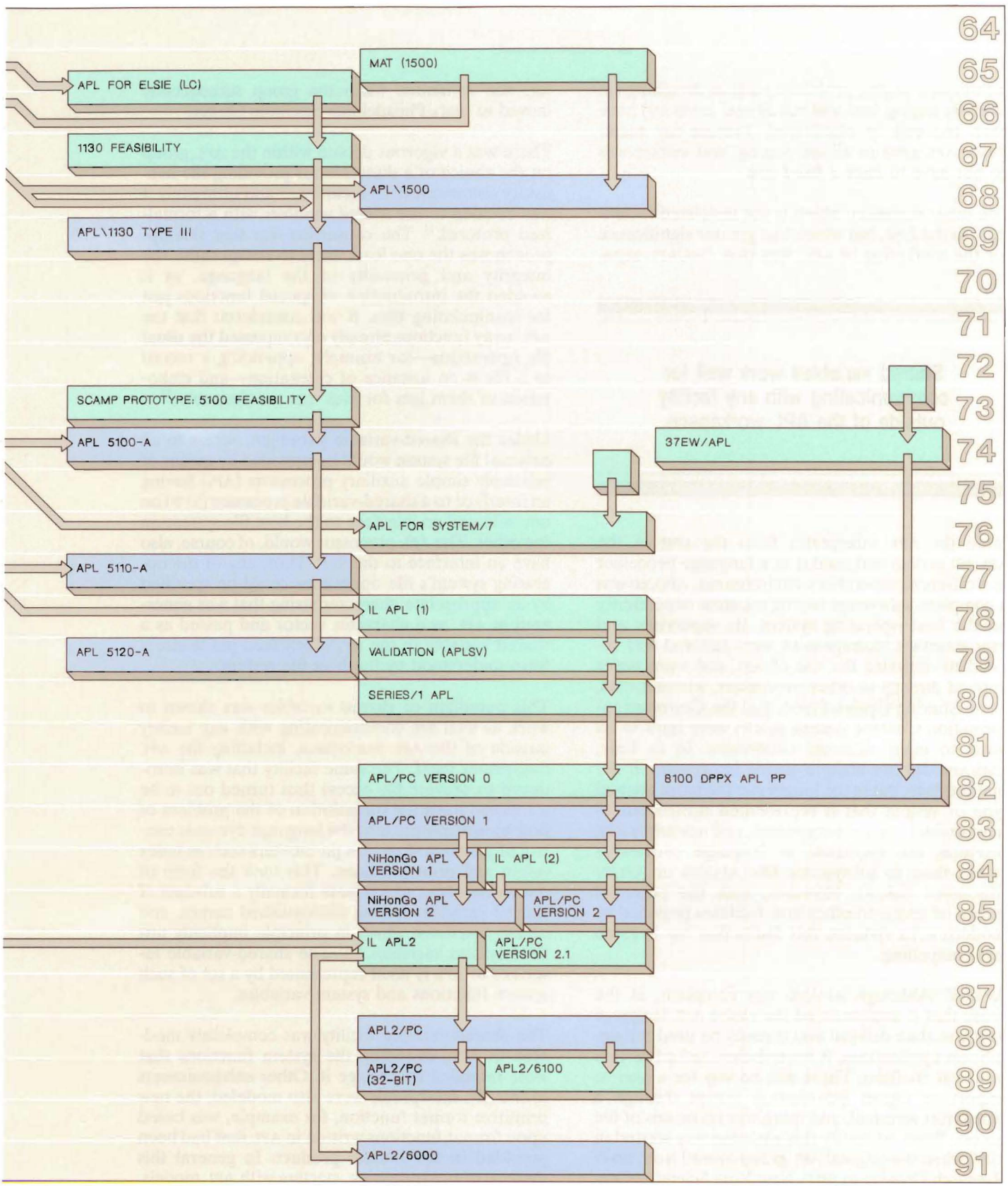
The first publicly available APL system was the cost-free "Type III" program (available without formal support) released in 1968. It was followed in 1969 by the two program products (PPs) shown in the chart. These were among the very first programs offered when IBM unbundled programs and hardware. An important decision taken then, which would influence the progress of APL in ways that even now are not completely understood, was to hold back the source code and release only object code to customers. This was done deliberately, to discourage proliferation of language variants and to give the original designers a better chance of directing the further evolution of APL along a coherent and consistent path. A positive effect of this policy was to facilitate formal standardization of APL later on, and the ad hoc standardization that resulted from having a single control point simplified the development of other APL products along the way. A possible negative effect was the discouragement of interest in APL as a subject of university research.

CMS/APL. An early variant of APL\360 was produced in IBM's Cambridge Scientific Center, where pioneering work on virtual systems was in progress. A small team there[9] adapted the APL\360 DOS code to make use of virtual storage under the Conversational Monitor System (CMS), running in the specialized hardware of the IBM System/360 Model 67. This CMS/APL system, which was made available as IBM's first installed user program (IUP), was also the first to explore two significant variations in the design of APL systems.

One such variation had to do with workspace size, which, in APL\360, was fixed at a constant value (of 32K bytes) for all workspaces in the system. By means of a relatively small modification to the internal structure of the workspace, CMS/APL enabled operation in the memory paging environment of the control program of CMS (CP/CMS) and enabled the use of variable-sized workspaces up to the capacity of the virtual storage available. An issue here, which was to be argued at length for a long time after, was the difference between swapping complete workspaces (in effect, paging logical units), and the paging of fixed segments of memory having no necessary relationship to the computational process occurring. It is probably fair to say that with the state of the art then, and for some time thereafter, swapping was more efficient, although it did require a uniform, fixed workspace size in the system. With modern hardware and programming

Figure 1 IBM APL systems




64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
MAT (1500)
APL FOR ELSIE (LC)
1130 FEASIBILITY
APL\1500
APL\1130 TYPE III
SCAMP PROTOTYPE: 5100 FEASIBILITY
APL 5100-A
37EW/APL
APL FOR SYSTEM/7
APL 5110-A
IL APL (1)
APL 5120-A
VALIDATION (APLSV)
SERIES/1 APL
APL/PC VERSION 0
8100 DPPX APL PP
APL/PC VERSION 1
NiHonGo APL VERSION 1
IL APL (2)
NiHonGo APL VERSION 2
APL/PC VERSION 2
IL APL2
APL/PC VERSION 2.1
APL2/PC
APL2/PC (32-BIT)
APL2/6100
APL2/6000

techniques, paging problems such as thrashing (inefficient paging into and out of real memory) have been reduced or eliminated. Present-day mainframe APL systems all use paging, and workspaces do not have to have a fixed size.

The other variation, which is not unrelated technically to the first, but which had greater significance for the marketing of APL, was that CMS/APL separated the APL interpreter from the rest of the APL\360 system and used it as a language processor in a different supervisory environment. APL\360 was a complete subsystem having minimal dependency on the host operating system. Its supervisor and user interface management were tailored and refined to optimize the use of APL and were never applied directly to other processors, whereas CMS, Time-Sharing Option (TSO), and the Customer Information Control System (CICS) were built to be hosts to many different processors. In its time, CMS/APL did not make a strong impression in the marketplace, but in the longer run the more general type of system that it represented turned out to have greater market acceptance, and nowadays APL products are marketed as language processors rather than as subsystems like APL\360 or APLSV (discussed below). However, with the powerful means of access to other host facilities provided by modern APL2 systems, this distinction has become less compelling.

Shared variables work well for communicating with any facility outside of the APL workspace.

APLSV. Although APL\360 was complete, in the sense that it implemented the entire APL language as it was then defined and it could be used for significant applications, it nevertheless lacked certain practical facilities. There was no way for a user to import or export information except through a typewriter terminal, and there was no means of file access. Work to rectify this situation was started in 1969 when the original APL group moved from IBM's Research Division to IBM's New York Scientific Center, and continued when the group subsequently moved to IBM's Philadelphia Scientific Center.

There was a vigorous debate within the APL group on the choice of a direction for providing the necessary communication facilities,[10] and ultimately it was decided to use *shared variables* with a formalized protocol.[11] The consensus was that this approach was the one least likely to compromise the integrity and generality of the language, as it avoided the introduction of special functions just for manipulating files. It was considered that the APL array functions already encompassed the usual file operations—for example, appending a record to a file is an instance of catenation—and elaboration of them just for files was not desirable.[12]

Under the shared-variable paradigm, access to an external file system would be provided by means of relatively simple auxiliary processors (APs) having an interface to a shared-variable processor (SVP) on one side and an interface to the host file system on the other. The APL processor would, of course, also have an interface to the SVP. Thus, any of the operating system's file operations could be specified by an appropriate character string that was generated in APL as a character vector and passed as a shared variable to the AP, which then put it into a form understood by the host file system.

This paradigm of shared variables was shown to work as well for communicating with any facility outside of the APL workspace, including the APL interpreter itself. The same facility that was introduced to provide file access thus turned out to be a rational basis for the solution of the problem of how to incorporate into the language dynamic control of primitive-function parameters such as index origin and print precision. This took the form of *system variables*, which were formally a subclass of shared variables having distinguished names, and *system functions*, which in principle implicitly utilized system variables.[13,14] The shared-variable interface to APL is itself represented by a set of such system functions and system variables.

The shared-variable facility was completely modeled in APL, including the system functions that were intended to manage it. Other enhancements to the APL interpreter were also modeled; the new primitive format function, for example, was based upon format functions written in APL that had been provided in the APL\360 product. In general this method of programming, starting with APL models,

was a multistage process. A functionally correct model was first written without regard to machine considerations, and when this was deemed to be correct, another version was produced using only APL primitives that could be easily mapped to machine code. Since both versions were executable, it was not too difficult to validate their functional equivalence, after which the second version could be used as a model for the final machine language program.

Experimental APLSV systems were produced for the then current System/360 operating systems in 1971 and 1972, as shown in Figure 1. Again, the job was accomplished in a relatively short time by a small, highly motivated team. An internal IBM announcement and a technical seminar on APLSV and shared variables was held in 1971, after which the Philadelphia Scientific Center made available on-line APLSV service to other IBM locations. This service was well received, and the high rate of usage constituted very effective testing for the product offering, which was made publicly available in 1973 in the form of a specially priced and contracted product, or programming request for price quotation (PRPQ).

The APL standard. Although questions were raised at the time, particularly in response to the seminar in 1971, regarding the wisdom of the shared-variable approach—as contrasted, for example, with building specific file and input/output facilities into the language—it does appear in retrospect that it was the proper direction. At the very least, by establishing a clear boundary between the language and the system facilities, it ultimately made it easier for the industry to agree on an APL standard. And by the same token, it has made it easier to build new APL systems, and to port APL systems between machines with dissimilar architectures.

The first official IBM standard for APL, put in place as an interim document in 1974, was the language as defined by the APLSV implementation.[15] Work on a formally written standard had already been started in the Philadelphia Scientific Center, but was still a long way from completion and adoption. Over the course of several years and many iterations, the work product and the responsibility was transferred to IBM's Santa Teresa Laboratory in California. Finally, after undergoing the formal ratification process in IBM, this formal document became the IBM APL standard in late 1977.[16] In 1979 the technical portion of this standard was published in its entirety as an appendix to a paper describing its evolution.[17] This appendix was later adopted as the first draft APL standard by a committee of the International Organization for Standards (ISO). It was not accepted as wholeheartedly by the American National Standards Institute (ANSI) committee, which insisted on rewriting the document in a different style altogether. Nonetheless, the APL language definition finally embodied in the standard adopted by all parties in 1987 is essentially that of APLSV.

Internal APLSV systems. By the time that the Philadelphia Scientific Center closed in mid-1974, IBM in general, and certain key sites in particular, had developed a strong dependency upon the APLSV service for running daily business. By this time also, the product direction had taken a turn, as discussed later, and there was not yet a fully supported APL product that could sustain the necessary maintenance and service level required. The affected sites therefore banded together to form an internal APL support group for the purpose of maintaining the APLSV program while they waited for a product to which they could satisfactorily migrate.

Some language development was included in the work of the support group, but their major activity was more in the nature of systems work—keeping up with evolving operating systems, and developing new or enhanced auxiliary processors for file management and other purposes. Notable among the latter was a processor, AP19, that enabled one active user to activate another user account under program control from inside the first user's workspace.[18] The first version of this worked only in a single machine, but a later version worked between machines not even necessarily in the same location. The primary motivation for this facility was the practical need to run long jobs in batch mode unattended, but it also made it possible to easily model and simulate general forms of cooperative and parallel processing.

APL/CMS and VS APL. While the original APL group was working on the design and development of APLSV in Philadelphia, a rather different line of inquiry was going on in IBM's Palo Alto Scientific Center in California. Here, the interest was in performance and the possibilities inherent in building a hardware APL machine. As shown in the first column of Figure 1, this work first resulted in a microcoded APL system for the System/360 Model 25. This was a single-user dedicated APL system in

which the control code that emulated the System/360 was replaced with code that emulated APL.[19]

APL\360 was used as the model of how an APL machine should appear to a user, and some pieces of code were used from existing systems, but overall the implementation was basically new. It introduced the use of arithmetic progression vectors (APV), which conserved both time and storage in many common situations, and facilitated more efficient evaluation of certain array transformations;[20] it made use of a very fast syntax analyzer that required a new internal representation of APL statements; and it used a different storage allocation method. Not all of APL was implemented at the microcode level, but this being an APL machine, the part not so implemented was necessarily written in the subset of APL that was microcoded. The supervisor program was also written in APL and executed that way without further translation.

The next step along this line of development was APL microcode for the System/370* Model 145. By this time (1972) APLSV had seen heavy use internally, and the shared variable concept had been generally accepted as the proper direction for managing system-related operations in APL systems. This technology was transferred, and other aspects of the work planned for the Model 145 were discussed, at a week-long workshop set up by the teams from Palo Alto and Philadelphia.

Also by this time, CMS as a time-sharing host was gaining in market acceptance, and a decision was taken by the Palo Alto group not to make a dedicated APL machine, as was done for the Model 25. Instead, they concentrated on an APL interpreter that would run under CMS and optionally use microcode to enhance its performance.[21] Two product offerings came directly out of this work: the interpreter with microcode assist, which could run only on the System/370 Model 145, and an independent interpreter named APL/CMS, which could run on any machine running CMS.

The microcode assist did indeed provide customers with a significantly more powerful APL processor than the Model 145 could provide without it, but its marketing was hampered by the fact that there was no similar upgrade available for the more powerful machines in the System/370 family. Although the design of the APL assist was quite general, the code itself could not be ported to other machines because they had a different underlying processor or did not use microcode at all.

While this work was going on in the Scientific Centers, plans were being made in the IBM Programming Center in Palo Alto for a new interactive time-sharing system to be called Virtual Systems Personal Computing (VSPC), and a principal language processor under that system was to be APL. Because of the marketing considerations noted previously in the discussions of CMS/APL and APL/CMS, this type of general time-sharing system, with independent language processors, was preferred over integrated systems like APL\360 or APLSV. As a consequence, when APLSV was made available as a product in 1973, it was given the more tentative status of a PRPQ, rather than full program product status, and the stand-alone interpreter developed in Palo Alto to run under CMS was chosen as the base for VS APL, the processor planned for VSPC. However, as an interim product of the type anticipated, the APL/CMS interpreter produced in the Palo Alto Scientific Center was also released then as a PRPQ.

In its original form and before it was actually put on the market, the APL/CMS interpreter had incorporated some language changes in addition to the changes in the internal design. Several of these were considered to cause problems in the language definition, and were opposed by the APL group in the Philadelphia Scientific Center where, as described earlier, work on an APL standard was already under way. The disagreement was escalated and resolved expeditiously under pressure of the need to get on with product plans. In addition to settling the issues of the moment, this resolution of the problem had the beneficial effect of accelerating the adoption of an APL standard within IBM, which, as noted earlier, has been an important factor in the continuing high productivity of APL development groups.

Eventually, the VS APL interpreter was produced by the APL product development group in the General Products Division of IBM as their first major product. They had previously (while still part of the Systems Development Division) taken over maintenance of APLSV when the Philadelphia Scientific Center closed in mid-1974. Over the course of the next several years, as shown in Figure 1, successive releases of VS APL added support for additional IBM mainframe time-sharing environments until all four—CMS, VSPC, CICS, and TSO—were included. A still extant final release was made in 1983.

An ongoing use of VS APL is the hands-on network environment (HONE) system, where APL has long been the vehicle for delivering configurators and financial analysis programs to the IBM marketing and support teams. This use posed two system problems that were not addressed by the APL product systems until the most recent release of APL2, described below. These problems arise in a situation in which large numbers of people must use identical programs but also maintain individual workspaces to hold their own data. First, if each person copies the programs into an individual workspace, and then saves it, the file storage system will be flooded with redundant material. Second, the common programs change over time as new products and new plans evolve. This information, which comes from centralized responsible sources, would somehow have to be propagated to all the copies in the individual workspaces.

The HONE solution to these problems was to develop a system facility where the individual users are given only *use access* to the common programs, which are held in a privileged storage area. The parties responsible for maintaining the programs can then upgrade as necessary the single copy held in common.

APL2. The evolution of APL2 is an interesting illustration of how a small group of people with a shared vision can maintain the continuity of their technical work and bring it to a successful conclusion, even over a time span of more than 15 years. During this time, people were transferred between three or four divisions and made several cross-country moves, all while producing other results of value to the company.

Thus, the desirability of breaking out of the constraints of rectangular arrays was recognized very early in the course of the work on APL, and some background work on the subject was steadily maintained in the Research Division while APL\360 was being developed. The group was then transferred to the Philadelphia Scientific Center, where definitive work, leading to an implementation of some form of generalized arrays, was started after the APLSV program was well along. When the center was closed in 1974, most of the APL group was transferred, as a group, to the West Coast, where they became part of the APL development organization. The work on a new APL interpreter—dubbed "APL2" at this point—was kept going there for a while, along with maintenance of APLSV, but the pressures of producing the VS APL products eventually reduced this to a crawl. However, language studies had been continued by the small contingent of the Philadelphia group that had remained on the East Coast, and the design of a new interpreter was resumed in earnest in 1978 after they and others were reassigned to the Research Division in Yorktown Heights, New York. The transfer of APL2 technology was completed later (1982), when the people directly working on the interpreter were again transferred to the APL development group in California.

The evolution of APL2 illustrates how a small group with a shared vision can be successful.

In keeping with the usual method of doing things in the APL development milieu, the initial work on APL2 did not start as a blank slate, but as a variation of the working APLSV interpreter. Actual coding started in Philadelphia in 1971, a comprehensive paper on the principal ideas was published in 1973,[22] and by 1974 an interpreter with general array operations was available for experimentation, first running under APLSV in the Philadelphia system, later running in Palo Alto, and later still in IBM Kingston, New York, as an alternative interpreter on their APLSV service system. As this evolved, new functions unrelated to general arrays were picked up from the APLSV internal releases.

The first APL2 product was an interpreter running under CMS, which was announced as being somewhat experimental and was marketed as an installed user program (IUP). In addition to the functions necessary for the accommodation of general arrays, it incorporated numerous language enhancements. These ranged from simply making the primitive mathematical functions work with complex numbers, through several new and extended primitive functions such as eigenvalues, picture format, and replication, to simple-sounding but far-reaching changes in APL operators, which were now able to accept defined functions as operands, and could themselves be user-defined.[23,24]

The APL2 IUP included an important new system function, ⎕TF, which either generated a transfer form—a system-independent representation—of an APL object, or established an object in a workspace from the transfer form. It also included two new system commands,)*OUT* and)*IN*, which generated and accepted host system files composed of collections of objects in transfer form. Although the primary motivation for these operations was to facilitate migration between different APL systems, in time these collections of APL objects in transfer form have come to be regarded as another form of saved workspace with its own useful characteristics, even where migration is not an issue.

A full-fledged APL2 program product, which emphasized system facilities for integration with other IBM programs as much as new language features, was released in 1984. The code was a further development of the IUP, with some emphasis on speeding up execution, some language changes, and a full complement of auxiliary processors. Many of these were inherited from VS APL, with or without enhancements. This use of existing code was facilitated by resolving some differences between APLSV and VS APL in the internal design of the shared variable processor to ensure portability of existing auxiliary processors. Notable among these were a full-screen session manager and a processor for access to database products such as DATABASE 2* (DB2*) and System Query Language/Data System (SQL/DS*). Communication with APL2 from the Interactive System Productivity Facility (ISPF) products was provided by an auxiliary processor distributed with ISPF. Other system facilities included national language support for system commands and messages, a new internal character type of four bytes per character for supporting large character sets such as Kanji, and various utilities to facilitate migration from older APL systems.

Carried over from the APL IUP was the use of *primitive defined functions*—functions written in APL rather than machine language that are nonetheless part of the language processor and are invoked by the use of primitive function symbols or system commands. First used to facilitate experimentation with language changes, primitive defined functions have been retained in the later releases of APL2, where they are used for a variety of system operations and primitive functions, or portions of primitive functions, for which high performance is not a requirement. There is also a complementary facility in APL2 that uses ordinary user-type names to invoke machine coded functions. This is a device that goes back to the first version of APL\360, where it was used to provide useful functions, variously called keyword functions or workspace functions, for which special-character names were not available. In the case of APL2 it was used for the eigenvalue and polynomial functions that were included as primitives in the IUP but were felt to be somewhat premature for inclusion as such in the program product.

The second release of APL2, which followed the first by little more than a year, continued the trend toward closer integration of APL with its environment. There were improvements in the support for database products and graphic display devices, and direct access was provided to system editors outside of APL. Of possibly greater significance, however, was the introduction of a new facility known as *name association*, where routines written in FORTRAN, assembler, or Restructured Extended Executor (REXX) could be called from APL applications.[25] This facility works by providing dynamic linking between the active workspace and other namespaces, allowing different parts of a process to be sequentially executed by different processors, as may be appropriate. Although inspired in part by a shared variable auxiliary processor developed many years earlier at the IBM Heidelberg Scientific Center in Germany,[26] it differs from the use of the shared variable facility in that the parts of the process are never executed in parallel or asynchronously, the associated names may refer to external objects of any kind (not just variables), and the name association is preserved across working sessions.

The third release of APL2, in late 1987, included two major extensions to APL2 system capabilities. One was the automatic utilization of hardware vector processing when available, an obvious exploitation of the natural array properties of APL. The other was the inclusion of an encapsulation mechanism for APL workspaces, which transformed them into load modules, known as *packages*, which could then be accessed by a name association processor. Among other applications, packages have the potential to solve the problems addressed by *use access* on the HONE APL system previously mentioned. The existing primitive defined function facility, which already depended upon isolation of namespaces for its operation, was used as an integral part of the implementation of the package facility. The associated processor was also extended

to support FORTRAN function calls in addition to calls to subroutines; and a complementary facility was provided to allow routines written in other languages to request execution of APL expressions.

In recognition of the greater availability of personal computers and workstations with versatile displays, and their use as terminals and for running native APL systems, this release of APL2 allowed the use of lowercase alphabetics as an alternative to underscored alphabetics, and provided a system command for setting the mode.

In earlier times of APL design and development there was a strong effort made to reach consensus on new ideas, and an equally strong emphasis on the importance of testing by users. As the development center shifted about and the development process itself became more formalized this was not lost sight of, although some aspects of it have been hard to maintain. Since about 1982, however, with the popularization of electronic conferencing, the IBM internal computer network has been used quite effectively to gather together user experience with developmental systems, and publicize opinions on new ideas. User testing of new systems has been formalized at the same time, with selected sites within IBM undertaking responsibilities as virtual extensions of the regular development test group.

Small machines

The first implementation of an APL-like system on a small machine was the PAT system on the IBM Model 1620, done in 1964. APL has had a presence of small machines ever since. In fact, as is detailed below, the first portable desktop personal computer marketed by IBM was designed as an APL machine.

APL\1130. In 1965–1966 the IBM Los Gatos Laboratory in California was working on the design of a very small, low-cost (hence LC or "Elsie") machine. It was to have a relatively simple instruction set and an internal memory of only 1024 words, supplemented by an external magnetic disk, about eight inches in diameter, which used grooves on one side for mechanically indexing to the magnetic tracks. Science Research Associates, then a subsidiary of IBM, was interested in the educational potential of such a machine, and commissioned a study to produce an APL system for it. Two of the three people who conducted the study had previously worked on IVSYS.[27] Drawing on this experience, the group proposed a modified architecture for Elsie, better suited to implementing APL. An emulator for this machine design, and an assembler for programming it, were written for the IBM Model 7090, and design of the APL system proceeded from there. The result was then successfully transferred to a real Elsie prototype, so that in due course an APL system was running in Los Gatos.

Unfortunately, business considerations kept Elsie from ever becoming a product, but the work on it was not wasted. By 1967 APL\360 was becoming widely known within IBM, and the Research APL group was approached by an IBM branch office interested in the possibility of having an APL system available for the IBM Model 1130, a midsize "scientific" machine. To quickly produce a prototype and show feasibility, an Elsie emulator was written for the Model 1130 and the APL system was installed on it. It ran successfully. To improve performance, one additional instruction was added to the Elsie emulator, an escape to the native 1130 architecture, which was used as the path to more efficient coding of successive parts of the interpreter. As shown in Figure 1, an upgraded APL\1130 was later produced as an IBM Type III program.

Not shown in the figure is a more formal APL\1130 product that had a very short life. It was a time-sharing upgrade of the Type III program, produced by the APL development group in Palo Alto, which was then still part of the Systems Development Division. It was shipped to one or two customers before being withdrawn from the market. But it, too, was not wasted. Indeed, it figured importantly in the early development of the modern personal computer.

APL 5100. In late 1972 the Palo Alto Scientific Center was asked by IBM's General Systems Division headquarters in Atlanta, Georgia, to suggest an APL product suitable for production by their division. In response, the Scientific Center proposed an entry-level machine that could fit on a desk. This suggestion was accepted, and they proceeded to assemble a team composed of people with hardware knowledge from Los Gatos and people with software knowledge from the Scientific Center to work on the design. The team selected a processor engine known internally as "Palm" for the machine's central processing unit, in preference to another, called UC.5, that was also available at the time.

Once again, the quickest way to show feasibility and produce a prototype was to emulate an existing ma-

chine that already had APL programmed for it. In this case, the Model 1130 was chosen. Thus, APL\1130, a system that had its origins in Elsie, the earlier Los Gatos machine, and that had been ported by emulation to the Model 1130, where it was eventually converted to native 1130 architecture code, was now ported to a new machine in which Los Gatos was also involved in the hardware design. The functioning prototype, know as SCAMP (Special Computer APL Machine Portable), was produced in the short time of six months, and was successful in persuading the General Systems Division to proceed with a production machine.[28]

At present the SCAMP prototype, an APL machine that was the unique forerunner of the first production personal computer, resides in the collection of the Smithsonian Institution in Washington, D.C.[29]

The production machine was designed at IBM's General Systems Division laboratory at Rochester, Minnesota, and was made available as a product, the IBM 5100 machine, in 1974—less than a year and a half from the start. This remarkably short development cycle for such a complex new product can be attributed in large part to the fact that emulation was used again, even in the final product. This time, however, although the same Palm internal engine was used, System/360 architecture was emulated rather than 1130 architecture, so that the up-to-date APLSV product system could be used as the APL facility with virtually no modification. There were some changes, however, that anticipated later developments in personal computers. For example, the primary input/output device was a cathode ray tube with an associated keyboard that included an extra shift, named "CMD," and a number pad; there was a software switch to enter a communication mode to enable the machine to act as a terminal on a host system; and another switch to automatically copy input and output to an attached printer.

The later models, the IBM 5110 and 5120, which had a different internal processing engine and also used a later version of APLSV, carried these forward-looking changes considerably further. Where the IBM 5100 had only a tape cartridge for nonvolatile storage of files and workspaces, the later machines included an eight-inch diskette facility, separately available in the IBM 5110 and integral in the IBM 5120. Whereas the CMD key in the IBM 5100 was used very modestly to generate APL system commands from six keys in the top row, the IBM 5110/5120 CMD key was also used to produce the APL overstrike characters, as well as the distinguished names of system variables and system functions, with a single shifted keystroke. The CMD key was also used to switch the entire keyboard from an APL character mode to a standard lowercase and uppercase character mode in which the single APL characters were still available as a third shift. All the models had a shared variable facility for communicating with the tape drive and the printer, and in the later models this was extended to include the diskette drives, the display screen, and the serial input/output port.

The SCAMP prototype, an APL machine, resides in the collection of the Smithsonian Institution.

There is considerable family resemblance between these early APL machines and the personal computer (PC) line of machines IBM produced a few years later. The IBM Portable Personal Computer, in particular, with its built-in small screen looks a lot like the IBM 5110, and its part number of 5155 is clearly in the sequence of the earlier machines. (The early PC itself is model number 5150, and the PC/XT* and PC/AT* have model numbers 5160 and 5170.) This is not really surprising, since the IBM Rochester development group that produced the 5100 and 5120 machines was later transferred to the IBM laboratory at Boca Raton, Florida, where they constituted the beginning of the Entry Systems Division of IBM, which developed the IBM PC.

APL\1500. Returning for a moment to the 1960s, a variant of the IBM 1130 machine was the IBM 1500, a system intended for the educational market. This system used a faster version of the 1130 processor, known as the 1800. The IBM 1500 was an early example of a multimedia machine, featuring a cathode ray tube display and a film projection unit in addition to the usual typewriter input and output. In 1965 the Service Bureau Corporation wrote a program called MAT/1500 for the IBM 1500, whose primary software was a computer-aided instruction program called "Coursewriter." MAT/1500 was in-

tended to augment this mostly verbal system with a mathematical capability, including elementary functions and some array operations.

Some three years later, Science Research Associates undertook to write a full APL system for the IBM 1500. They modeled their system after APL\360, which had by that time been developed and seen substantial use inside of IBM, using code borrowed from MAT/1500 where possible. It is interesting to note that in their documentation they acknowledge their gratitude to "a number of high school students for their compulsion to bomb the system."[30] This was an early example of a kind of sportive, but very effective, debugging that was often repeated in the evolution of APL systems.

DPPX APL. At about the same time that the Palo Alto Scientific Center was working on SCAMP, another APL system design was under way at IBM in Poughkeepsie, New York, using the UC.5 engine that had been considered as an alternative to Palm when Palo Alto selected its processor engine. When nearly completed, the project was moved to Kingston and the target machine became the IBM 8100, which had the UC1 as its internal engine, an upgrade of the UC.5. This was to have been a complete APL system, including its own supervisor, but work on it was halted before it reached product level. The project was subsequently moved again, this time to the Lidingo laboratory of IBM Sweden. The technology transfer was effected in part by the temporary assignment of one, and then another, of the original developers. It was brought to product status running under the Distributed Processing Program Executive (DPPX) operating system of the IBM 8100, rather than its own supervisor.

DPPX APL was a multiuser time-sharing system that made innovative use of the shared variable processor in its internal operations. (Work on its design also led to suggestions for broadening the functionality of shared variables, which, though not implemented at the time, are still worth considering.[31]) Motivated by an absolute limit of 64K bytes for the workspace size, the designers consigned as much function as possible to the shared variable processor, so as to free up space in the workspace that would otherwise be taken up with the interfaces to other parts of the system. Thus, for example, communication to the keyboard and display input and output was mediated by the same shared variable processor as was available at the user level. Also, to facilitate the use of shared variables between workspaces—a means of overcoming the workspace size limitation as well as a way of functionally segmenting programs—the system provided support functions to start and control secondary sessions from inside an active workspace, much in the manner of the AP19 processor on the internal APLSV systems described earlier.

The system emphasized the utilization of DPPX facilities from inside APL programs. Sets of support functions, which had the same appearance as the workspace functions mentioned previously in the discussion of APL2, were provided, for example, to facilitate the use of the DPPX Presentation Services (PS). Alternatively, these operations, and others, could be effected by means of explicit shared variables using an auxiliary processor connecting directly to DPPX input/output and command programs. This gave the APL programmer willing to work at that level access to the operating system commands and macros.

Another innovation, at the APL language level (which was otherwise essentially that of VS APL), was the introduction of a system variable, □CMD, to which a character string depicting an APL system command could be assigned. Thus, it was possible to imbed in a running program an order to save the workspace at that point, while the program continued to run. Though sometimes controversial, this feature of dynamic execution of system commands was well thought out, as were the other innovations in DPPX APL. It is unfortunate that the system did not see enough real use for a body of opinion to build upon the value of these innovations. Still, there is little doubt that with its emphasis on communication and integration with the environment, DPPX APL was a step in the right direction, as evidenced by subsequent developments in the two major current APL systems, APL2 and the derivatives of IL APL, discussed next.

IL APL. In 1974 the Computer Science Department of the IBM Madrid Scientific Center started an APL system for the IBM System/7, a small sensor-based machine intended for use in applications such as process control and laboratory automation. The APL system was modeled after APLSV in the expectation that the use of shared variables would simplify both the design and the subsequent operation of the sensor input/output, but the APLSV code itself was not used. In order to accommodate an APL time-sharing system to a machine that had as little as 16K of two-byte words in its main memory, the

interpreter was modularized so that its parts could be swapped into memory much the same way as the workspaces. The system was coded in assembly language.[32]

System/7 APL was never made into an IBM product, but it saw some use in several research laboratories both inside and outside of IBM, and was used by the Madrid Scientific Center itself to control the environment in an experimental agricultural project. Its major significance, perhaps, was that it was the first implementation of APL by a team that went on to develop a portable APL system that has been the basis for the IBM implementations of APL on personal computers and workstations.

In 1976 the Scientific Center was asked to write an APL system for the IBM Series/1*, the successor to the IBM System/7. Reluctant to simply repeat the same work in another low-level language, the team conceived the idea of writing a portable APL system in a systems programming language intermediate between assembler and a high-level language such as APL. The language they designed, known as IL (for Intermediate Language), has a simple syntax, somewhat resembling APL, and a semantics closely related to that of assembly languages, but tailored to the requirements of an APL system.[33] An APL system written in this language can be ported to different machines by writing compilers from IL to each. Since each compilation is essentially a one-time affair, the execution speed of the compilers is not an issue, but the time to produce one is, and therefore they have been written in APL.[34]

The IL approach was first tested by writing an interpreter only, and compiling it to System/370, where it could be compared to APLSV and debugged. Once this was successful, the IL implementation was expanded to include an APL system command handler, an input editor and scanner, and a shared variable processor.[35] Nearly all of the coding for IL APL was new, taking only a few algorithms from APLSV and VS APL. Others were based on publications, some of which were also the source for APLSV and other mainframe APL systems.[36,37]

Series/1 APL. After the validation of IL APL on the IBM System/370, the first download porting was to the Series/1. It was still necessary to code machine-dependent parts of the system, such as the APL time-sharing supervisor and library management operations, by other means. The IL interpreter was also modified for the Series/1. The architecture of this machine placed severe limitations on the size of the APL workspace, and to mitigate this problem the IL APL designers developed the idea of a two-part workspace: a *main workspace* of the maximum size, where APL objects were created and modified, and an *elastic workspace*, which used a secondary memory to swap out APL objects not currently referenced, when more execution space was needed.

A choice had to be made between two operating systems on Series/1: Realtime Programming System (RPS), which was the official IBM offering, and Event-Driven Executive (EDX), which was then being developed informally by interested groups in the company. The Madrid Scientific Center did not have resources to do both machine-dependent subsystems. RPS was selected, on the basis that it was the mainline offering, while internal interest in an APL system on EDX was probably strong enough to generate its own separate support. In fact, this proved to be the case, and a support group for an EDX version was formed under the aegis of the APL Design Group in Research. A viable EDX system was produced,[38] which was used in about 40 internal IBM sites. Neither version was ever offered as an IBM product.

APL/PC. The second download porting of IL APL was to the IBM Personal Computer (PC), in 1982.[39] One requirement placed on the design was that it should be usable in a PC with only 128K of random access memory, a configuration that was considered generous at that point in the evolution of the personal computer market. But even with larger memories, in order to achieve acceptable performance it was necessary that the workspace size stay within the 64K primary addressing capability of the machine. To reduce the severity of this limitation, the elastic workspace concept was carried forward from the Series/1 design.

The language level of APL/PC was essentially that of the APLSV internal system, which included picture format, ambivalent defined functions, and the execute alternative system function. All of these were also in the APL2 IUP, which became available at about the same time as the zero-level of APL/PC, but not in VS APL, the principal mainframe product at the time. APL/PC also included ⎕TF and `)IN` and `)OUT`, as found in the APL2 IUP. In addition to facilitating communication and migration between different APL systems, especially between mainframes and PCs, the use of the transfer form also

served to overcome the absence of the APL copy command in APL/PC.

An important aspect of the design of APL for the PC was the deliberate effort made to bring as much of the underlying machine as possible under control of the APL programmer. This took two forms. First, a new system function, □PK, was introduced to allow access to any part of the machine memory for both reading and writing, and to execute machine-code subroutines. Second, auxiliary processors were provided to interface with the Basic Input/Output System (BIOS) and Disk Operating System (DOS) interrupts, with the DOS file system, and with peripheral devices, including the display.

APL2 supports 32-bit addressing for the PS/2 and runs on the AIX platform for the IBM RISC System/6000.

The development versions of APL/PC were tested by the APL Design Group in Research, using scripts and programs first constructed in connection with work on APL2. A preproduct-level program was then made available for testing by interested parties in many different parts of the company, before the first product offering was released in 1983. This was the beginning of an iterative process—upgrading or changing the IL APL, subjecting the resulting PC program to widespread internal use and testing, and product release—a process that is still going on, through several versions of APL/PC, APL2/PC, and APL2 for workstations.

The next use of IL APL was the porting to the IBM 5550, the personal computer available in Japan, done in collaboration with the IBM Tokyo Scientific Center. This resulted in a product known as NiHonGo APL. For this version the internal data types were expanded to include two-byte characters, and the keyboard and display operations were elaborated, so as to accommodate the much larger Kanji character set. Otherwise, NiHonGo APL and APL/PC were the same.

In the period from 1984 to 1986, a second IL APL interpreter was developed and also ported to the IBM 5550 machine. The main changes affected memory management, and many of the implementation limits of the first version were markedly increased. There were also some performance improvements, and a substantial increase in the number and scope of the auxiliary processors. Most significant among the latter was AP2, an interface to non-APL programs, which made it possible to dynamically load and run DOS programs or programs written in FORTRAN or assembly language. This processor was under development at about the same time as the name association facility in APL2, and represents an alternative approach to solving the same problems.

There was one more refinement of APL/PC, a version intended for internal use only, which included support for IBM Personal System/2* (PS/2*) Model 80, and a workspace packaging program. Although the same term is used, the resulting APL/PC package is quite different from that of the mainframe APL2. In this case, a separate program, running directly in DOS, uses the name of an APL workspace and the list of auxiliary processors it uses, and produces a DOS (.EXE) program that contains the workspace and the necessary parts of the APL system and can therefore run independently.

APL2/PC. Over the period from 1986 to 1990, an IL implementation of APL2 was produced, and successively enhanced, by the Madrid Scientific Center in collaboration with the IBM United Kingdom Scientific Centre in Winchester.[40] There have been two releases of this system and several field upgrades. The first release, in 1988, was a 16-bit version that can run on any of the IBM PC or PS/2 machines, and requires only 256K of real memory. It retains most of the implementation limits of APL/PC version 2, which derive from the 16-bit addressing structure of the underlying structure, but the workspace size can be as large as 440K bytes. Except for complex number arithmetic and some minor language refinements, it is a full-function APL2 system with a comprehensive set of auxiliary processors, a full screen manager modeled after the mainframe APL2 version, and direct invocation of DOS operations by means of a *)HOST* system command.

The 32-bit version, released in 1989, was generated, downloaded, tested, and debugged in 13 man-weeks, an impressive confirmation of the effectiveness of the IL approach. In this version there is no

practical limit on workspace size, which can be as large as 15 megabytes, for example, on a 16-megabyte PS/2, and there are no separate limits on the size of APL variables. It has all of the language and system features of the 16-bit version, and both may be used to produce running packages of APL applications, as described previously.

APL2/6000. The most objective test of the IL APL approach was the most recent one, the porting to an Advanced Interactive Executive* (AIX*) platform on the IBM RISC System/6000*. In this case, one person with no prior knowledge of either IL APL or the RISC System/6000—working alone except for a few days of help at the end—was able to produce the necessary back end of the IL compiler, which translates the IL code to the language of the object machine, and bring up a viable APL workspace on the machine in less than 10 weeks. With a second person writing the machine-dependent parts of the program in C, the system was brought to the point of being publicly demonstrated less than six months from the start. An internal IBM release was reached in 10 months and a product announcement was made two months after that.

Other APL processors

All of the APL machine implementations described so far (and shown in Figure 1) are interpreters, as befits the language processor in a highly interactive system. However, there has been a steady pressure in the marketplace to improve the performance of production applications in APL. As a result, in addition to the microcode assist described above, acceleration techniques ranging from adaptive interpretation, to translation to intermediate languages, to direct compilation to machine language have been worked on and used experimentally.

An adaptive interpreter for APL was designed in the IBM Israel Scientific Center in the mid-'70s. The program analysis was implemented in APL, and it compiled code to an intermediate language conceived of as a virtual APL machine.[41] The implementation was completed far enough to estimate its performance, which was promising as far as it went, but no production use was made of it. However, the techniques were further evaluated in the APL Design Group in the Research Division when one of the investigators took an assignment there, and they provided background for the APL compiler work that followed.

This compiler work branched into two principal directions, both of which used APL itself as the principal programming tool. One direction emphasized the exploitation of APL array operations to directly generate very fast machine code and take advantage of the potential for automatic parallelization of APL programs at the basic block level.[42] At first relatively narrow in the range of APL expressions it could compile, this program has been improved and enhanced to the point where other internal IBM sites are experimenting with it for production applications while the investigation continues in the Research Division. Consideration is currently being given to translating into another high-level language, rather than directly into machine code.

The other branch of the Research Division work in APL compilers started out with the intent to translate into a high-level language, namely FORTRAN, in order to take advantage of the optimizing compilers already extant for that language and the portability implied by the widespread availability of FORTRAN compilers.[43] The general scheme of this compiler is to work within the APL2 system, compiling those functions in an application that are most resource consuming, and invoking the compiled functions at run time by means of the name association facility in APL2. An important objective of the work on this compiler was to translate all of APL, up to its chosen language level, without compromising on the nuances of end conditions or other detailed aspects of the language definition. The work was transferred to the numerically intensive computing (NIC) group at IBM in Kingston, New York, around 1987, where it underwent enhancement of its user interface and was migrated from CMS to Multiple Virtual Storage. Finally, under the aegis of that group, the program, now known as AOC (APL2 Optimizing Compiler), was turned over to an IBM Business Partner for marketing and further enhancement. It was announced as a product in early 1991.[44]

Another instance of translating APL to a high-level language is the work done in the IBM Federal Sector Division using Ada as the target language. The translator was written in APL2, and had the limited goal of allowing an algorithm designer to prototype rapidly in APL and, after debugging there, translate the program to Ada for compilation and running in that environment.[45] The APL acceptable to this translator had to be highly stylized, but it nevertheless turned out to be useful in an important prototyping application.

Concluding remarks

It is perhaps fitting to make note of some of the things not discussed in this paper. Foremost among these is all the work on APL implementation done outside of IBM. The actual number of implementations of APL is in the dozens, most of which have, or have had, an economic life. Virtually every major manufacturer of computers has had its own implementations, starting very early in the history of the language, and many of these, like the systems produced or modified by APL time-sharing vendors, have contributed to the evolution of the language itself.

As noted in the text, APL has figured prominently in the evolution of small machines. Its very interactive nature, combined with the simplicity and power of its array operations, has been a magnet for designers of small machines. Thus, even before the IBM 5100 was developed, a small Canadian company, Micro Computer Machines, had built several APL machines small enough to fit in an attache case. At the present time, there are implementations for all the major families of small computers, as well as for several workstations and lesser-known small and intermediate machines.

Another large area untouched by this paper is that of applications written in APL, except for one. That one, of course, is the design and implementation of APL systems. As the APL compilers come into their own, this field of application may well broaden significantly.

Finally, it should be mentioned that there has been an unbroken series of international APL conferences since 1969, and numerous implementers' workshops and standards committee meetings, at which language, implementation, and standardization issues have been refined to the benefit of all concerned. Thus, IBM's family of APL systems has evolved in an active and stimulating environment that continues to attract the kind of highly talented people who made it happen in the first place.

Acknowledgments

While I have tried to achieve a dispassionate and even-handed tone in describing the developments in APL products, the actual events often took place in a far more emotional, and sometimes adversarial, atmosphere where points of view were advanced with fervor and fiercely defended. In the course of writing this paper I consulted with many of the people involved in these events, most of whom are mentioned in the references. Without exception, the responses were not only helpful, but warm with the remembrance of past associations. I hesitate to list their names, for fear I may inadvertently leave some out, but I want to thank them all for their present help, and for their earlier contributions to the evolution of APL systems. I also want to express my gratitude to John C. McPherson, whose name does not appear elsewhere herein, but whose influence was pervasive. Now a retired IBM vice president, John recognized the value of APL very early, and shared his technical insights and gave support and encouragement to everyone involved in APL development throughout the course of the work.

Cited references and notes

1. Figure 1 is an extension and elaboration of one produced by R. H. Lathwell in 1982.
2. H. Hellerman, "Experimental Personalized Array Translator System," *Communications of the ACM* **7**, 433 (July 1964).
3. A. D. Falkoff and K. E. Iverson, "The Evolution of APL," in *History of Programming Languages*, H. L. Wexelblat, Editor, Academic Press, New York (1981), p. 666.
4. H. A. Kinslow, "The Time-Sharing Monitor System," *Proceedings AFIPS 1964*, FJCC **26**, Spartan Books, Washington DC (1964), pp. 443–454.
5. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962).
6. A. D. Falkoff and K. E. Iverson, "The APL\360 Terminal System" in *Interactive Systems for Experimental Applied Mathematics*, Academic Press, New York (1968).
7. L. M. Breed and R. H. Lathwell, "The Implementation of APL\360," in *Interactive Systems for Experimental Applied Mathematics*, Academic Press, New York (1968).
8. The Grace Murray Hopper Award of the ACM, presented to L. M. Breed, R. H. Lathwell, and R. E. Moore in 1973. L. J. Woodrum of the IBM Poughkeepsie Laboratory contributed code for sorting and other operations, and provided continuing assistance in the development of APL\360.
9. W. Barrett and M. F. C. Crick.
10. A. D. Falkoff, "A Survey of Experimental APL File and I/O Systems in IBM," *Colloque APL*, Institut de Recherche d'Informatique, Rocquencourt, France (1972).
11. R. H. Lathwell, "System Formulation and APL Shared Variables," *IBM Journal of Research and Development* **17**, No. 4, 353–359 (1973).
12. The technology and the concept have now come together, 20 years later. As this paper was going to press, Release 1 of APL2 version 2 was announced, one of its principal new features being the ability to directly apply primitive APL functions to host system files.
13. A. D. Falkoff and K. E. Iverson, *Communication in APL Systems*, Technical Report 320-3022, IBM Philadelphia Scientific Center, PA (1973).
14. A. D. Falkoff and K. E. Iverson, "The Design of APL," *IBM Journal of Research and Development* **17**, No. 4, 324–334 (1973).

15. IBM Corporate Bulletin C-B 3-9045-001 (October, 1974).
16. IBM Corporate Standard C-S 3-9045-001 (December, 1977).
17. A. D. Falkoff and D. L. Orth, "Development of an APL Standard," *APL79 Conference Proceedings, APL Quote Quad* **9**, No. 4, Part 2, 409–453, ACM, New York (1979).
18. B. J. Hartigan, "AP19—A Shared Variable Terminal Interface for APL Systems," *APL81 Conference Proceedings, APL Quote Quad* **12**, No. 1, 137–141, ACM, New York (1981).
19. A. Hassitt, J. W. Lageschulte, and L. E. Lyon, "Implementation of a High Level Language Machine," *Communications of the ACM* **16**, No. 4, 199–212 (1973).
20. A. Hassitt and L. E. Lyon, "Efficient Evaluation of Array Subscripts of Arrays," *IBM Journal of Research and Development* **16**, No. 1, 45–47 (1972).
21. A. Hassitt and L. E. Lyon, "An APL Emulator on System/370," *IBM Systems Journal* **15**, No. 4, 358–378 (1976).
22. Z. Ghandour and J. Mezei, "General Arrays, Operators and Functions," *IBM Journal of Research and Development* **17**, No. 4, 335–352 (1973).
23. D. A. Rabenhorst, "APL2 Language Manual," SB21-3015, IBM Corporation (1982); available through IBM branch offices.
24. J. A. Brown, *The Principles of APL2*, Technical Report 03.247, IBM Santa Teresa Laboratory, CA (1984).
25. J. A. Brown, J. Gerth, and M. Wheatley, *Communication Method Between an Interactive Language Processor and External Processes*, U.S. Patent No. 4,736,321 (1988).
26. H. Eberle and H. Schmutz, *Calling PL/I or FORTRAN Subroutines Dynamically from VS APL*, Technical Report 77.11.007, IBM Heidelberg Scientific Center, Germany (1977).
27. L. M. Breed and P. S. Abrams; the third person was W. S. Worley, Jr.
28. P. J. Friedl, "SCAMP: The Missing Link in the PC's Past?," *PC* **2**, No. 6, 190–197 (November 1983).
29. J. Littman, "The First Portable Computer," *PC World* **1**, No. 10, 294–300 (October 1983).
30. S. E. Krueger and T. D. McMurchie, *A Programming Language\1500*, Science Research Associates, Chicago, IL (1968).
31. K. Soop and R. A. Davis II, "Extended Shared-Variable Sessions," *APL85 Conference Proceedings, APL Quote Quad* **15**, No. 4, 148–150, ACM, New York (1985).
32. M. Alfonseca, M. L. Tavera, and R. Casajuana, "An APL Interpreter and System for a Small Computer," *IBM Systems Journal* **16**, No. 1, 18–40 (1977).
33. M. L. Tavera and M. Alfonseca, *IL. An Intermediate Systems Programming Language*, Technical Report 01-78, IBM Madrid Scientific Center, Spain (1978).
34. M. Alfonseca and M. L. Tavera, "A Machine-Independent APL Interpreter," *IBM Journal of Research and Development* **22**, No. 4, 413–421 (1978).
35. M. L. Tavera and M. Alfonseca, *The IAPL Machine-Independent APL Processor*, Technical Report 03-80, IBM Madrid Scientific Center, Spain (1980).
36. R. H. Lathwell and J. E. Mezei, *A Formal Description of APL*, Technical Report 320-3008, IBM Philadelphia Scientific Center, PA (1971).
37. A. D. Falkoff, "A Pictorial Format Function for Patterning Decorated Numeric Arrays," *APL81 Conference Proceedings, APL Quote Quad* **12**, No. 1, 101–106, ACM, New York (1981).
38. P. A. McCharen, *The Series 1 APL-EDX System Installation and User's Guide*, Technical Report 19.0552, IBM Burlington, VT (1981).
39. M. L. Tavera, M. Alfonseca, and J. Rojas, "An APL System for the IBM Personal Computer," *IBM Systems Journal* **24**, No. 1, 61–70 (1985).
40. M. Alfonseca and D. A. Selby, "APL2 and PS/2: The Language, the Systems, the Peripherals," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 4, 1–5, ACM, New York (1989).
41. H. J. Saal and Z. Weiss, "A Software High Performance APL Interpreter," *APL79 Conference Proceedings, APL Quote Quad* **8**, No. 4, 74–81, ACM, New York (1979).
42. W.-M. Ching, "Automatic Parallelization of APL Programs," *APL90 Conference Proceedings, APL Quote Quad* **20**, No. 4, 76–80, ACM, New York (1990).
43. G. C. Driscoll, Jr. and D. L. Orth, "Compiling APL: The Yorktown APL Translator," *IBM Journal of Research and Development* **30**, No. 6, 583–593 (1986).
44. "1991: The Year of the APL2 Optimizing Compiler," *Interlink, January 1991*, Interprocess Systems, Inc., Atlanta, GA (1991).
45. J. G. Rudd and E. M. Klementis, "APL-to-Ada Translator," *APL87 Conference Proceedings, APL Quote Quad* **17**, No. 4, 269–283, ACM, New York (1987).

Accepted for publication June 27, 1991.

Adin D. Falkoff *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.*
Mr. Falkoff is currently a research staff member in the Computer Science Department at the Thomas J. Watson Research Center. He joined IBM in 1955, and since 1960 has worked on various aspects of computer science, including APL. He was a member of the visiting faculty at the IBM Systems Research Institute for several years, and a visiting lecturer in computer science at Yale University. From 1970 to 1974, Mr. Falkoff established and managed the IBM Philadelphia Scientific Center, and from 1977 to 1987 was the manager of the APL design group at the Thomas J. Watson Research Center. He received a B.Ch.E. from the City College of New York in 1941 and an M.A. in mathematics from Yale University in 1963, the latter under the IBM Resident Scholarship Program. He has received IBM Outstanding Contribution Awards for the development of APL and the development of APL\360.

Reprint Order No. G321-5443.

APL2: Getting started

by J. A. Brown
H. P. Crowder

APL is a concise and economical notation for expressing computational algorithms and procedures. This paper introduces the main ideas of APL2, an IBM implementation of APL, and illustrates the programming style with some graphical examples.

Originally developed as a mathematical tool for teaching computer concepts, APL offers a systematic and structured method for thinking about computational problems and implementing solutions. Because the APL notation can be executed directly on computers, APL is a rich and powerful programming language, suitable for solving a wide range of computational problems in science, engineering, and business.

The original APL notation was described by Iverson in 1962.[1] The first commercial computer programming implementation of the language was documented in 1968,[2] and in 1971, Brown extended the APL notation in his work at Syracuse University.[3] APL2, the IBM implementation of extended APL, is documented in Reference 4 and today is used as a problem-solving tool for a wide variety of applications, as one may conclude reading papers such as those described in References 5–8.

APL2 consists of three fundamental components: arrays, functions, and operators. *Arrays* are the data structures of APL2, consisting of collections of numbers and text characters. *Functions* are programs that manipulate arrays; functions take arrays as arguments and produce new arrays as results. *Operators*, a powerful concept in APL2, are programs that manipulate functions; they take functions as operands and produce new functions as results.

The purpose of this paper is to introduce the key APL2 concepts of arrays, functions, and operators and how they relate and interact in a unique problem-solving environment. Several examples are provided that show how solutions to some interesting problems can be expressed precisely and concisely.

APL2 arrays

Arrays are the data structures of APL2. Arrays are collections of data, the values being numbers or characters or both. Arrays have structure and are organized as single elements, vectors, matrices, and higher-dimensional rectangular arrangements. In addition, APL2 arrays can be structured in hierarchical arrangements, as described later.

Arrays can have names that are used to refer to their contents in APL2 expressions. Shown below are arrays named *A* (containing a single number), *NUMS* (containing a list of five numbers), *CHAR* (containing a matrix of six characters), and *MIXED* (containing both numbers and characters):

```
      A
3

      NUMS
1 3 5 7 3.14159

      CHAR
CAT
FAT

      MIXED
2 BE OR NOT 2 BE
```

These examples show how the values of arrays are displayed by APL2; input is indented from the left margin, and output is flush left. The default display shows the array values but little about the array structure. To better understand the structure of APL2 arrays, use the function *DISPLAY* to construct pictures that show array structure. Following is *DISPLAY* applied to the previous examples:

```
      DISPLAY A
3

      DISPLAY NUMS
┌─────────────┐
│1 3 5 7 3.14159│
└─────────────┘

      DISPLAY CHAR
┌→┐
↓CAT│
 FAT│
└─┘

      DISPLAY MIXED
┌→──────────────────┐
│ 2 ┌→──────┐ 2 ┌→┐ │
│   │BE OR NOT│   │BE│ │
└∈──└───────┘   └─┘ │
```

In the first example, the array *A* is displayed with no structural information. In APL2 terms, *A* is a *simple scalar*; it has only value and no structure of interest. In the next example, *NUMS* is displayed in a box with an arrow on the top edge, indicating that *NUMS* is a vector or one-dimensional array. The matrix *CHAR* is displayed in a box with two arrows, indicating that the data are arranged along two dimensions. Finally, the display of *MIXED* indicates that it is a vector containing both simple scalar numbers (two instances of the number 2) and two character vectors.

In the last example, *MIXED* is an instance of a *nested array* that has other arrays as items. The following sequence builds up and displays a more complicated nested array *D*:

```
      A←2 2⍴ 10 11 12 (13 14)
      B←15
      C←16 17 18
      D←A B C

      DISPLAY D
┌→──────────────────────┐
↓ ┌→──────┐ 15 ┌→──────┐ │
  │10  11 │    │16 17 18│ │
  │12 ┌→┐ │    └───────┘ │
  │   │13 14│ │          │
  └∈──└─┘   │ │          │
└∈──────────┘ │          │
```

The array *D* is a vector with three items. The first item is a two-by-two matrix, one of whose items is again a vector of length two. The second item of *D* is the simple scalar 15, and the third item is a vector with three items.

APL2 arrays are very powerful but simple in concept. An APL2 array is a rectangular arrangement of items; any item in the array can be a single number, a single character, or another array of arbitrary complexity. This ability to structure data as nested APL2 arrays offers two major benefits. First, most data processing and computational data structures can be modeled and captured as APL2 arrays and thus used in APL2 applications. And second, as demonstrated in following sections, complicated APL2 data structures allow simpler APL2 application programs that are easier to design, code, and maintain.

As a final example of an array, the nested array *SALESDATA* is a matrix having four rows and five columns. *DISPLAY* shows all the detail of the matrix and each item in row one and column one is a character vector; every other item is a single number:

```
      DISPLAY SALESDATA
┌→┌→──────────┬→──┬→──┬→──┬→──┐
↓ │REGION/QTR │1Q │2Q │3Q │4Q │
  ├→──────────┼→──┼→──┼→──┼→──┤
  │NORTHEAST  │632│1256│959│1033│
  ├→──────────┼→──┼→──┼→──┼→──┤
  │MID-COAST  │719│548 │1179│1180│
  ├→──────────┼→──┼→──┼→──┼→──┤
  │SOUTHEAST  │1435│884│1020│1331│
  └→──────────┴→──┴→──┴→──┴→──┘
└∈
```

The default display of *SALESDATA* in APL2 has the following form similar in appearance to a spreadsheet report:

```
      SALESDATA
REGION/QTR    1Q    2Q    3Q    4Q
NORTHEAST    632  1256   959  1033
MID-COAST    719   548  1179  1180
SOUTHEAST   1435   884  1020  1331
```

This array structure is identical to the data aggregates that are created and manipulated by relational data systems. This ability for APL2 arrays to consistently represent data relations has resulted in APL2 being used for data analysis and manipulation in conjunction with relational database management systems.

APL2 functions

APL2 functions are programs that manipulate and perform calculations with arrays. Functions take arrays as their arguments and create new arrays as their results. In APL2, functions can be either *primitive* or *defined*. A third class of functions is discussed later. Primitive functions are part of the APL2 language and are provided with the APL2 Program Product from IBM. Defined functions are programs that are composed of primitive and defined functions. APL2 provides a rich set of primitive functions, but a subset of these is introduced here so that interesting examples can be presented.

In the previous section on arrays, the defined function *DISPLAY* is used to further understand the structure of APL2 arrays. *DISPLAY* takes as its argument any APL2 array and produces a character matrix showing the array's structure. The primitive function **reshape** denoted by the symbol "ρ" is also used to convert a list into a matrix. **Reshape** works on both numbers and characters:

```
      3ρ'A'
AAA

      2 2ρ1 2 3 4
1 2
3 4

vs      2 3ρ'CATFAT'
CAT
FAT
```

The same symbol is used for the function **shape** which yields information about the structure of its argument:

```
      ρ3ρ'A'
3

      C
CAT
FAT

      ρC
2 3
```

In APL2, for conservation of symbols, each symbol represents two functions. When the symbol is written with one argument (on the right) you get one function, and when the symbol is written with two arguments (one on each side) you get the other function. In most cases, the two functions are related. In the case of **shape**, the result is an array that gives structural information—the "shape"—about its array argument. In the case of the related function **reshape**, the result is an array whose structure is dictated by the left argument and is composed of items from its right argument.

An important concept in APL2 is the *rank* of an array—the number of directions along which data are arranged. The rank of an array is the number of items in the shape of the array, so it follows that rank is obtained by applying the **shape** function to the shape of an array. Matrices have rank 2 (data arranged in rows and columns), vectors have rank 1 (data arranged along one direction) and scalars have rank 0 (no structure; data arranged along zero directions). The following is an example of each:

```
      ρρ2 3ρ'CATFAT'
2
```

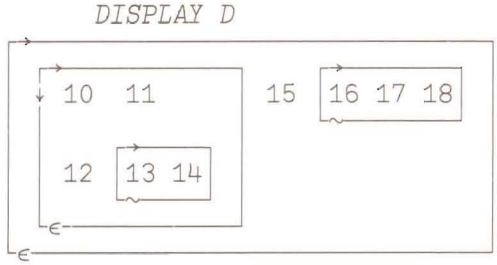
```
      ⍴⍴1 2 3
1

      ⍴⍴'A'
0
```

A large class of functions in APL2 is called *scalar functions* because the functions apply to the simple scalars of their arguments independent of array structure. Examples include most of the arithmetic functions such as **addition** (denoted by +), **subtraction** (−), **multiplication** (×), **division** (÷), **power** (⋆), **maximum** (⌈), **minimum** (⌊), and the scalar functions include the relational functions such as **less than** (<), **less than or equal** (≤), and equal (=). Some examples are:

```
      2 3 4 + 5 6 7
7 9 11

      1 2 3 ≤ 3 2 1
1 1 0

      100 × 1 2 3
100 200 300
```

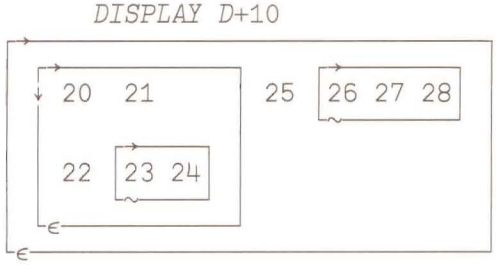
and in the example before of a nested array *D*, where:

```
      DISPLAY D
```



an arithmetic function example is:

```
      DISPLAY D+10
```



When a single item is presented to a scalar function, the scalar is paired with every item in the other argument. This powerful concept, *scalar extension*, is used frequently in following examples.

A useful function for array manipulation is **catenate** (denoted by ,). **Catenate** is used to join arrays to form new arrays:

```
      A
1 2 3

      B
100 200 300

      A,B
1 2 3 100 200 300

      M
1 2
3 4

      N
100 200 300
400 500 600

      M,N
1 2 100 200 300
3 4 400 500 600

      N,0
100 200 300 0
400 500 600 0
```

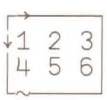
Interval (denoted by ⍳) produces arrays based on numerical sequences. **Interval** and arithmetic scalar functions can be combined to produce a wide variety of arrays:

```
      ⍳7
1 2 3 4 5 6 7

      2×⍳7
2 4 6 8 10 12 14

      ¯8+2×⍳7
¯6 ¯4 ¯2 0 2 4 6

      2⋆⍳7
2 4 8 16 32 64 128
```

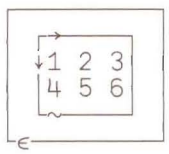
Notice that in APL2 expressions, functions are executed from right to left.

Enclose (denoted by ⊂) is used to convert any collection of data into a scalar. For example:

```
      DISPLAY Q
```



```
      DISPLAY ⊂Q
```

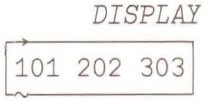


In the second expression above, the *DISPLAY* function shows the result to be a scalar. The data are organized along no axes and have rank 0. Inside the scalar, however, the complete original array is retained. Therefore **enclose** returns a scalar that contains its argument as its only item.

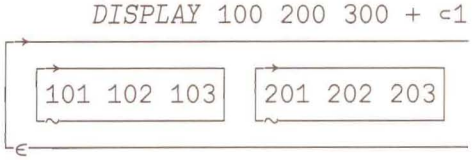
This data structure has several practical applications. Arrays are sometimes used in situations where the structure is not important. **Enclose** allows hiding the inner structure of arrays. For example, *'JIM'* is a three item character vector. If an application treats this array as a name then the fact that it has three items is not relevant. The expression ⊂*'JIM'* hides the structure, making it easier to treat it as a single object (a name).

Enclose is also useful if the contents of an array are required to participate in scalar extension. Note the difference that **enclose** makes in the following examples:

```
      DISPLAY 100 200 300 + 1 2 3
```




```
      DISPLAY 100 200 300 + ⊂1 2 3
```



In this second expression, the scalar ⊂1 2 3 is paired with each of the numbers 100, 200, and 300.

Defined functions in APL2 are programs that consist of a sequence of APL expressions. Syntactically, defined functions are used in the same manner as primitive functions. The function *AVG*, for example, computes the average of a list of numbers:

```
[0]    Z←AVG X
[1]    ⍝ COMPUTE THE NUMERIC AVERAGE
[2]    ⍝ OF VECTOR <X>
[3]    Z←(+/X)÷⍴X

       AVG 3 9 7 11 14
8.8

       2 × AVG 3 9 7 11 14
17.6
```

The function *SD* computes the standard deviation of a list of numbers; it invokes *AVG* as a subfunction:

```
[0]    Z←SD X
[1]    ⍝ COMPUTE THE NUMERIC
[2]    ⍝ STD DEVIATION OF VECTOR <X>
[3]    Z←AVG X
[4]    Z←((+/(X-Z)*2)÷⍴X)*.5

       SD 3 9 7 11 14
3.709
```

APL2 operators

APL2 operators take existing functions as arguments and produce new functions as results. The functions produced by operators are called *derived functions*. Operators can process both primitive and defined functions.

The operator **reduction** (denoted by /) takes a function as operand and produces a related derived function. Derived functions are the third class of functions. What follows is an example of **reduction** applying to the **addition** function producing the **summation** function and applied to the **maximum** function producing the **largest of** function:

```
       +/1 3 4 2 5
15

       ⌈/1 3 4 2 5
5
```

If *F* is any function, then the expression *F/A B C* is equivalent to the expression ⊂*A F B F C*.

Reduction also produces functions that operate on arrays of higher rank:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12

      +/M
10 26 42

      ⌈/M
4 8 12
```

The operator **each** (denoted by ¨) applies its function operand to each item of an array. For example, the **interval** function "⍳" can be combined with **each** to produce a derived function that produces arrays of arithmetic intervals:

```
      DISPLAY ⍳5
┌→────────┐
│1 2 3 4 5│
└─────────┘

      DISPLAY ⍳¨⍳5
┌→──────────────────────────┐
│ ┌→┐ ┌→──┐ ┌→────┐ ┌→──────┐ │
│ │1│ │1 2│ │1 2 3│ │1 2 3 4│ │
│ └─┘ └───┘ └─────┘ └───────┘ │
└∈──────────────────────────┘

┌→────────┐
│1 2 3 4 5│
└─────────┘
```

Each can produce derived functions that take two array arguments. For example, **each** applied to the **reshape** function "⍴" produces a function useful for building structured arrays:

```
      DISPLAY (⍳3) ⍴¨ 'ABC'
┌→──────────┐
│ ┌→┐ ┌→┐ ┌→──┐ │
│ │A│ │BB│ │CCC│ │
│ └─┘ └──┘ └───┘ │
└∈──────────┘

      DISPLAY 3 ⍴¨ 'ABC'
┌→──────────┐
│ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │AAA│ │BBB│ │CCC│ │
│ └───┘ └───┘ └───┘ │
└∈──────────┘
```

In the second expression above, the left argument 3 was replicated by scalar extension to apply to each item of the character vector right argument. Using **enclose** to produce scalars for scalar extension can give the following type of result:

```
      DISPLAY 2 3 ⍴ 'ABCDEF'
┌→──┐
↓ABC│
 DEF│
└──┘

      R←2 3 ⍴ 'ABCDEF'
      DISPLAY (⊂3 4) ⍴¨R
┌→──────────────────────────┐
↓ ┌→──┐ ┌→──┐ ┌→──┐          │
 ↓AAAA ↓BBBB ↓CCCC          │
  AAAA  BBBB  CCCC          │
  AAAA  BBBB  CCCC          │
 └───┘ └───┘ └───┘          │
 ┌→──┐ ┌→──┐ ┌→──┐          │
 ↓DDDD ↓EEEE ↓FFFF          │
  DDDD  EEEE  FFFF          │
  DDDD  EEEE  FFFF          │
 └───┘ └───┘ └───┘          │
└∈──────────────────────────┘
```

Each can apply to defined functions exactly as it applies to primitive functions. Next, the *AVG* function is applied to a vector of numeric vectors to produce a vector of averages:

```
      DISPLAY A
┌→──────────────┐
│ ┌→──────┐ ┌→──┐ │
│ │3 5 4 7 6│ │2 4│ │
│ └───────┘ └───┘ │
└∈──────────────┘

┌→──────────────────┐
│3.75 2.95 5.45 12.85│
└───────────────────┘

      AVG¨ A
5  3  6.25
```

This expression applies the program *AVG* over and over again to the items of data in *A*. This is close to the definition of iteration. The APL2 **each** operator is the array analogue of iteration. It permits the writing of many iterative computations without a loop.

APL2 examples

The following sections present three different examples that illustrate APL2 programming style. Use of APL2 is by no means restricted to these kinds of applications.

A graphical example of the each operator. Earlier it was seen that the **each** operator was useful for introducing structure into nested arrays. Here **each** is used at a higher level for drawing pictures.

The following APL2 defined function draws a circle on a graphics device:

```
[0]   DIAM CIRCLE LOC
[1]  ⍝ SIMPLE CIRCLE FUNCTION
[2]   'GSMOVE' GDMX LOC-.5×DIAM
[3]   'GSCOL' GDMX↑CLR_WHL←1⍜CLR_WHL
[4]   'GSARC' GDMX LOC,360
```

The left argument of *CIRCLE* is a single number, the diameter of the circle to be drawn. The right argument is a pair of numbers giving the *x-y* coordinate of the center of the circle. The function consists of calls to the *GDMX* function that is supplied with IBM's APL2 Program Product. *GDMX* uses the Graphical Data Display Manager (GDDM*)[9] to perform graphics primitives, but any graphics system could be used in a similar manner.[10]

As an example, an expression to draw a single circle using the *CIRCLE* function is:

```
2 CIRCLE 0 0
```

Figure 1A shows the resulting picture.

Consider now the requirement to draw several circles using the basic circle-drawing routine. In most programming languages, this would involve designing and writing a higher-level program to stage data for repetitive calls to *CIRCLE*. In APL2, this additional structure can be incorporated into the data instead of the program. For example, consider the following expression:

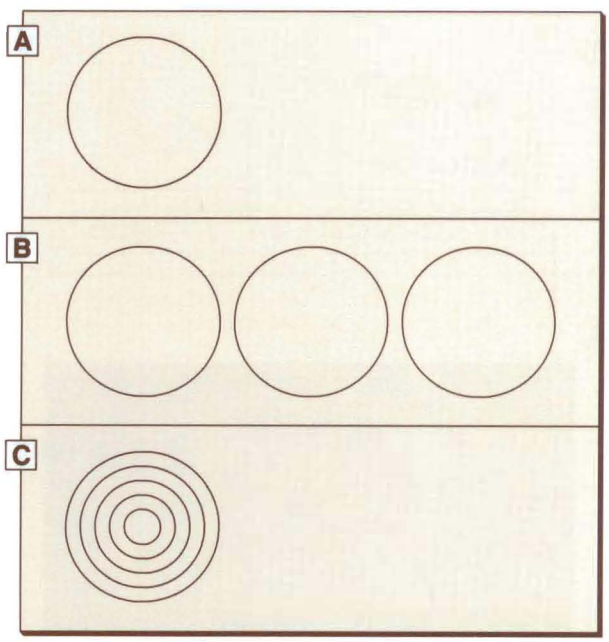
```
2 CIRCLE¨ ¯4 0 4,¨0
```

Figure 1B shows the graphical result of executing this expression.

In this example, we are using *CIRCLE* with the **each** operator. The resulting function is applied to the vector of pairs in its right argument (recall right to left execution). Since the left argument is a scalar number, all circles are drawn the same size.

In the next example, *CIRCLE* is used with a vector left argument of sizes and a scalar right argument indicating location:

Figure 1 Result of drawing one, three, and five circles




```
(⍳5) CIRCLE¨ ⊂0 0
```

The resulting arrangement of concentric circles is shown in Figure 1C.

The final example involves a more complicated calculation. Building the right argument to *CIRCLE* is similar to the example shown in Figure 1B—we are constructing a vector of pairs representing locations of multiple circles. The *y* coordinate of each pair is computed using 1○, the APL2 function for mathematical SINE. Figure 2A shows the result of *DISPLAY* on the first two pairs:

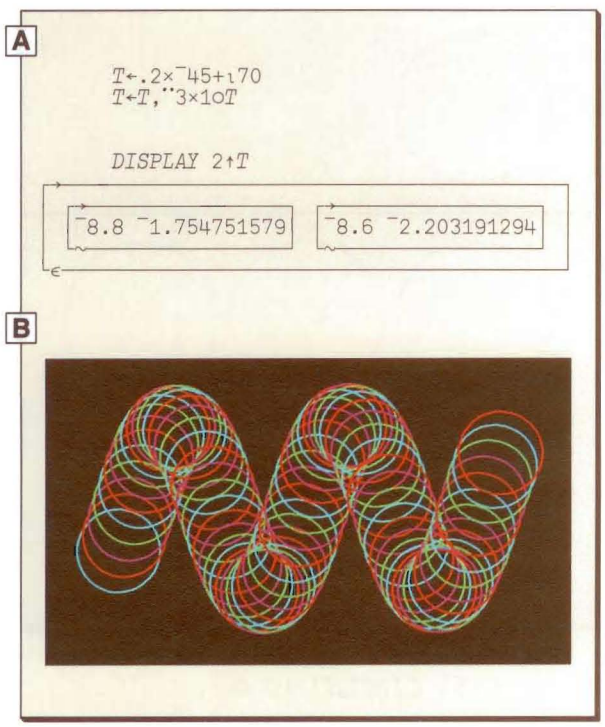
```
T←.2×¯45+⍳70
T←T,¨3×1○T
```

Figure 2B is the graphical result of executing the final expression:

```
1.5 CIRCLE¨ T
```

This example demonstrates the power of APL2 arrays. The use of hierarchical arrangement allows the representation of complicated data structures. But in addition, the structure of data arrays replaces the unnecessary complicating programming structure that clutters application programs; com-

Figure 2 Result of drawing multiple circles



plexity is moved out of programs and into the data. There is no explicit loop here; there is no IF... THEN ... ELSE. The structure is in the data, not in the program. This simplifies application design, implementation, and maintenance, and encourages modular design and program reuse.

Representing and manipulating sparse arrays. Many computational applications are required to create, manipulate, and process *sparse arrays* whose elements are mostly zero. It is wasteful in both memory and computation to process these data as full arrays. In many cases, especially for large arrays, structures can be used to encapsulate these data in a sparse format. APL2 does not have a built-in sparse array representation, but depending on the application and the nature of data manipulation and calculation required, sparse structures can be represented by APL2 nested arrays.

A sparse vector can be represented as a two-item nested array; the first item contains the indices of the nonzero coefficients in the vector, and the second item contains the coefficients themselves. For example, the vector *V* has most of its elements equal zero:

```
      V
0 0 0 0 4 0 0 0 0 0 0 2 0 0 0 3
```

The function *SVPACK* packs vectors into a sparse format:

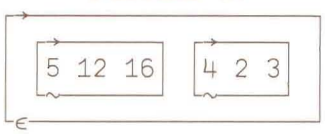
```
[0]    Z←SVPACK V;I
[1]   ⍝ PACKS A FULL VECTOR <V>
[2]   ⍝ INTO A SPARSE VECTOR <Z>
[3]    I←V≠0
[4]    Z←(I/⍳⍴V)(I/V)
```

Now the result of:

```
      SV←SVPACK V
```

is:

```
      DISPLAY SV
```



A common computational operation on arrays is inner product. The following example shows a function *SIP* performing an inner product between a full vector *FV* and a sparse vector *SV*. In APL2 terms, this should give the same result as the inner product of *FV* and the nonsparse representation of *SV*:

```
[0]    Z←V SIP S
[1]   ⍝ INNER PRODUCT OF
[2]   ⍝ FULL VECTOR <V>
[3]   ⍝ WITH SPARSE VECTOR <S>
[4]    Z←V[⍋S]+.×2⊃S

      FV
1 4 3 3 2 1 4 4 5 2 3 5 1 1 3 4

      FV SIP SV
30

      FV +.× V
30
```

A sparse matrix can be represented as a list, each item of which is a sparse vector representing a column of the matrix. The array *SM* represents a matrix with three rows and four columns:

```
      ⍴SM
4
```

```
      ⍴⍤⍤SM
2 2 2 2
```

Next the four items are arranged in a two-by-two matrix so that the result of *DISPLAY* fits on the page, as shown in Figure 3.

The function derived from *SIP* using **each** can be used to premultiply *SM* by a full vector:

```
      (⊂1 3 2) SIP⍤⍤ SM
7.3 7.8 13.5 7.2
```

This calculation should give the same result as performing the analogous calculation with the full matrix. In the next example, the function *UNPACK* restores sparse matrices to full two-dimensional APL2 matrices. Inner product on full arrays is performed by the derived function +.×:

```
      UNPACK SM
1.1 1.2 0   0
0   2.2 2.3 2.4
3.1 0   3.3 0

      1 3 2 +.× UNPACK SM
7.3 7.8 13.5 7.2
```

Simulation and analysis of dice throws. A data analysis example is discussed next, illustrating the functional programming style of APL2. In this mode of APL2 application design, a series of computational steps are each performed by separate functional units, with the result of one functional unit becoming the operand of the succeeding functional unit. Because functional units are independent, they can be "unplugged" and replaced by functionally equivalent units; this allows experimentation with various implementation strategies and fine-tuning of the application.

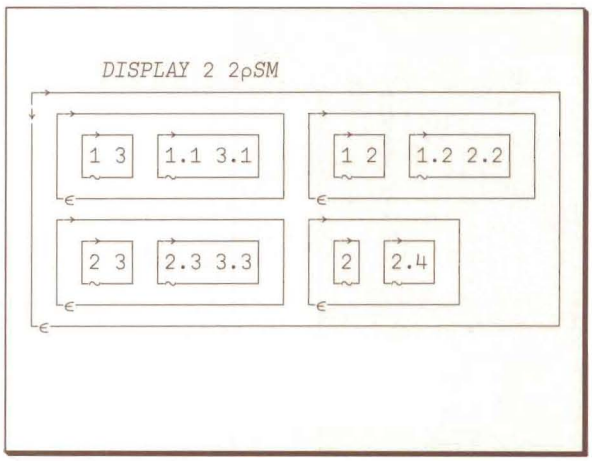
The function *DICE* is used to simulate a prescribed number of rolls of a pair of dice:

```
[0]   Z←DICE N
[1]  ⍝ ROLL DICE <N> TIMES
[2]   Z←?(N,2)⍴6
```

Now if the number of rolls of the dice are 5 and 8:

```
      DICE 5
4 1
1 4
2 4
5 2
5 6
```

Figure 3 A two-by-two matrix




```
      DICE 8
2 6
5 2
3 3
1 3
3 5
4 3
6 4
2 5
```

The argument *N* is the number of rolls to simulate. The result of executing *DICE* is an *N*-by-2 matrix, each row representing a dice roll. *DICE* uses the APL2 function **roll** (denoted by *?*), which produces random numbers. In this particular application, the elements of the result are picked from the pseudo-random uniform distribution in the range 1 to 6.

Next, the function *COUNT* can be used with *DICE* to summarize the results of a series of dice rolls:

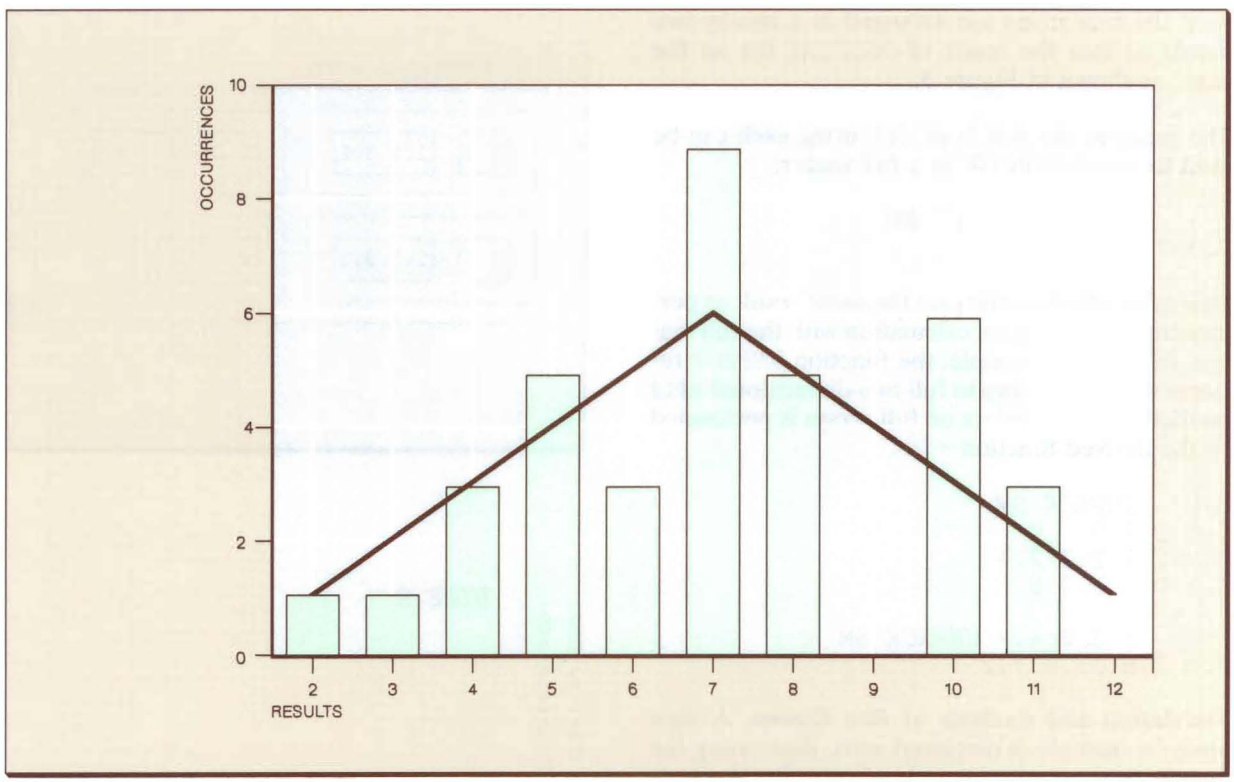
```
[0]   Z←COUNT A
[1]  ⍝ COUNTS DICE THROWS IN <A>
[2]   Z←+/A
[3]   Z←+/(1+⍳11)∘.=Z
```

Now the expression:

```
      A←DICE 7
```

results in:

Figure 4 Result of 36 dice throws




```
      A
2 5
2 4
2 5
6 3
1 1
4 2
5 2
```

The argument to *COUNT* is a dice-roll series produced by *DICE*. *COUNT* computes the sum of the two dice values for each roll, and tabulates the totals of each sum in the series. The result Z is an integer list of length 11; Z[1] contains the number of 2s rolled in the series, Z[2] contains the number of 3s, and so on. The sum of Z equals the number of rolls.

```
      COUNT A
1 0 0 0 2 3 0 1 0 0 0

      COUNT DICE 50
2 1 2 9 6 10 9 6 4 1 0

      COUNT DICE 500
14 33 37 50 65 89 73 42 58 22 17
```

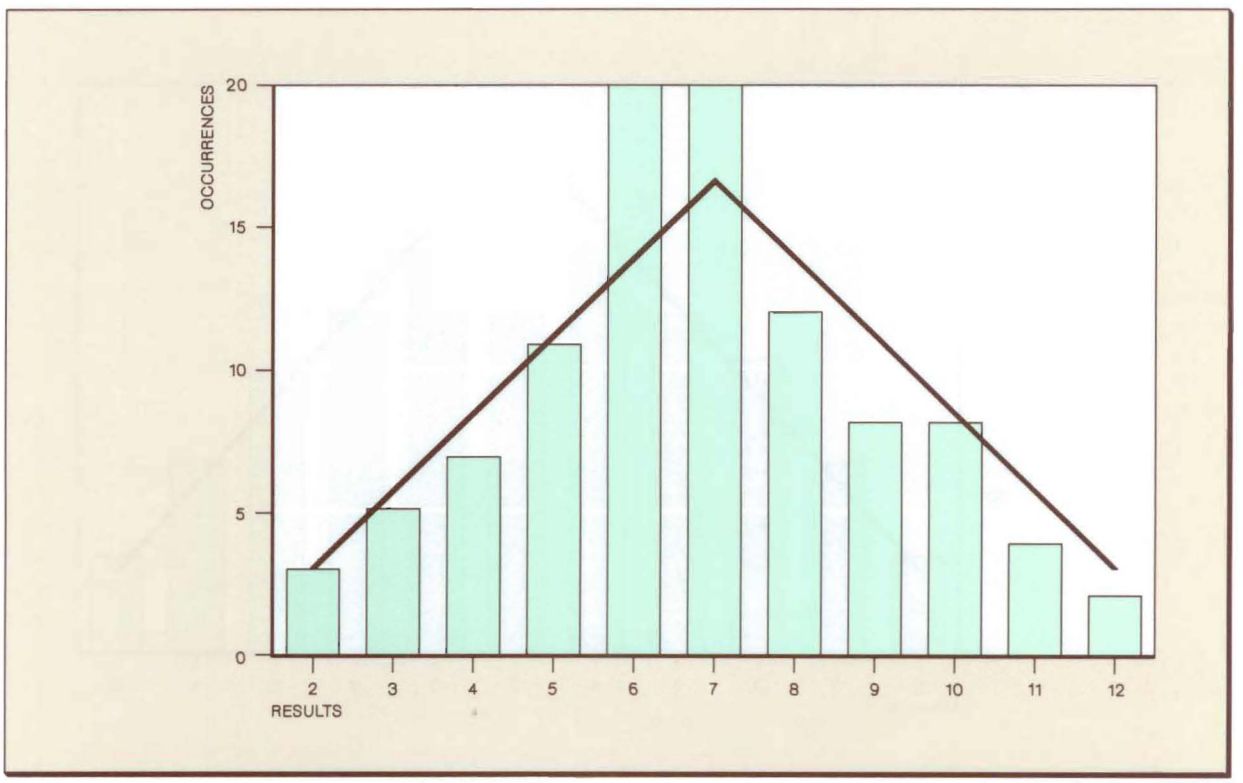
Continuing the discussion, the function *EXPECT* can be used to compute the expected number of dice-pair sums for a prescribed number of rolls:

```
[0]   Z←EXPECT N;T
[1]  ⍝ EXPECTED NUMBER OF EACH SUM
[2]  ⍝ FOR <N> DICE THROWS
[3]   Z←N×(T⌊⌽T←⍳11)÷36
```

The argument *EXPECT* is the number of dice rolls. The result Z is a list of length 11; Z[1] gives the number of expected occurrences of 2s in *N* rolls, Z[2] gives the number of expected occurrences of 3s, and so on. Some examples are:

```
      EXPECT 36
1 2 3 4 5 6 5 4 3 2 1

      EXPECT 72
2 4 6 8 10 12 10 8 6 4 2
```

Figure 5 Result of 100 dice throws




```
      EXPECT 500
13.9 27.8 41.7 55.6 69.4 83.3 69.4
                    55.6 41.7 27.8 13.9
```

Finally, the function *DRAW* can be used to plot the actual and expected results of a dice roll series. The main component of *DRAW* is the *CHARTX* function distributed with IBM's APL2 Program Product. *DRAW* accepts a two-item list. The first item is the actual results of dice-roll simulations as generated by *DICE* and *COUNT*; the second item is a list of expected dice-roll results as computed by *EXPECT*:

```
[0]   DRAW D;FORMNAME
[1]  ⍝ CHARTS ACTUAL AND EXPECTED
[2]  ⍝ DICE ROLLS
[3]   FORMNAME←'DICE'
[4]   (1+⍳11)CHARTX⊃D
```

The following expression simulates 36 dice throws and produces the picture in Figure 4:

```
DRAW (COUNT DICE 36) (EXPECT 36)
```

Figure 5 shows the result of the following expression with 100 dice throws.

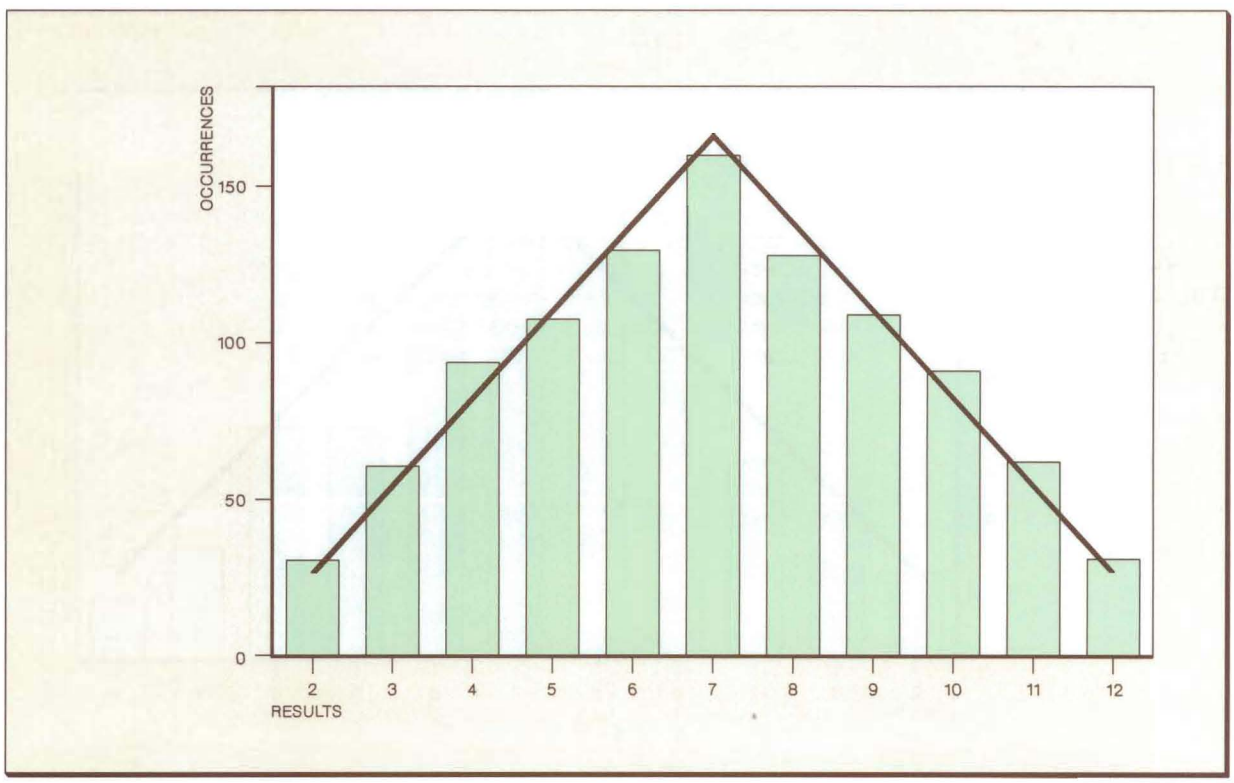
```
DRAW (COUNT DICE 100) (EXPECT 100)
```

Note that as the number of simulated rolls increases, the actual occurrences come closer proportionately to the expected occurrences, giving an empirical confirmation of the statistical law of large numbers. The absolute deviation of actual from expected grows as the number of rolls increases. The following expression simulates 1000 dice rolls and the result is shown in Figure 6:

```
DRAW (COUNT DICE 1000) (EXPECT 1000)
```

The functional programming style of APL2 encourages the construction of complicated programs from less complicated subprograms. This ability, derived from the APL2 language syntax, can result in shorter application development times and more error-free code. In addition, it can simplify application maintenance and encourage code reuse.

Figure 6 Result of 1000 dice throws



Conclusion

APL2 is one of the most powerful array processing notations in existence. But this power does not come only from the existence of structured data. Much more important is the ability of the structural data to control the flow of execution of a program. The structure of the data determines how algorithms are applied rather than determining the controls that the programmer inserts into a program.

This is why APL2 programs can be very small and easy to write and maintain. The complicated structure that sometimes permeates programs and makes them large and hard to manage is removed from the program and placed into the data, leaving programs that more accurately reflect the user's vision of the problem solution. APL2 is one alternative solution to structured programming.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references and note

1. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962).
2. A. D. Falkoff and K. E. Iverson, *APL\360: User's Manual*, IBM Corporation (1968).
3. J. A. Brown, *A Generalization of APL*, Ph.D. thesis, Department of Computer and Information Science, Syracuse University, Syracuse, NY (1971), Clearing House 74h004942 AD-770488./5.
4. *APL2 Programming: Language Reference*, SH20-9227, IBM Corporation (1988); available through IBM branch offices.
5. Stanley Jordan and Erik S. Friis, "The Foundations of Suitability of APL2 for Music," *IBM Systems Journal* **30,** No. 4, 513–526 (1991, this issue).
6. M. Alfonseca, "Advanced Applications of APL: Logic Programming, Neural Networks, and Hypertext," *IBM Systems Journal* **30,** No. 4, 543–553 (1991, this issue).
7. A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator," *IBM Systems Journal* **30,** No. 4, 527–538 (1991, this issue).
8. J. R. Jensen and K. A. Beaty, "Putting a New Face on APL2," *IBM Systems Journal* **30,** No. 4, 469–489 (1991, this issue).
9. *GDDM Version 2 General Information*, GC33-0319, IBM Corporation (1990); available through IBM branch offices.

10. The precise definition of this function is not relevant to the discussion; however, an explanation of what the function does follows: Line 1 is an APL2 comment. Line 2 puts the center where requested. Line 3 selects a color. In GDDM colors are indicated by integers. This line rotates a vector of integers and uses the leading one as the color of this circle. Each time the function is called, it chooses the next color in sequence. Line 4 draws an arc of 360 degrees (i.e., a circle).

Accepted for publication June 21, 1991.

James A. Brown *IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95150.* Dr. Brown is currently IBM's chief APL architect in the Technical Computing Solutions Department in Kingston, New York, and also in the APL Products Department in IBM's Santa Teresa Laboratory. He is responsible for the overall design of IBM APL systems and for marketing strategies. Dr. Brown received his Ph.D. in computer and engineering science from Syracuse University and his graduate thesis became the basis for the IBM APL2 products. He is a member of the Computer Science Accreditation Board that certifies computer science curricula at universities, and he is the language editor for the *ACM Quote Quad.*

Harlan P. Crowder *IBM Corporation, 1530 Page Mill Road, Palo Alto, California 94304.* Dr. Crowder is currently a consultant in the areas of application and technology with IBM's Technical Computing Systems Department. He is responsible for support and services for technical computing technology, including mathematical sciences, optimization, computer languages, and applications ranging from high performance computing to analytical business solutions. Dr. Crowder received a B.S. in chemistry from East Texas State University, an M.S. in operations research and industrial engineering from New York University, and a Ph.D. in computer science from the City University of New York. He is a member of the ACM, the Institute of Management Sciences, and the Operations Research Society of America.

Reprint Order No. G321-5444.

Extending the domain of APL

by M. T. Wheatley

This paper explores connectivity mechanisms between APL and other languages and applications available on a modern computer system. The design, implementation, and application of APL facilities such as shared variables, auxiliary processors, external names, file subsystems, and namespaces, as they are implemented in IBM's APL2 product, are discussed and compared.

Due to the persistence and insight of men like Iverson and Falkoff, in APL we are blessed with a language which, after more than 25 years of use, is still elegant, concise, precise, general, usable, and machine-independent.

> The definition of APL is purely abstract: the objects of the language, arrays of numbers and characters, are acted upon by the primitive functions in a manner independent of their representation and independent of any practical interpretation placed upon them. The advantages of such an abstract definition are that it makes the language truly machine independent, and avoids bias in favor of particular application areas.[1]

Despite the importance of machine-independence, a language that is used for computer programming cannot practically exist without access to the computing environment in which it runs. Further, to be useful in a wide variety of applications, such a language must also be able to access many of the other tools, libraries, routines, and subsystems available in that computing environment.

In the last 25 years, APL implementations have grown significantly in their ability to interact with the computing environment, including its associated software tools. This paper reviews the key facilities in APL that provide this function, briefly focusing on their history, objectives, characteristics, benefits, and problems. The discussion is centered around IBM implementations of APL.

Description of facilities

Early APL systems. When APL was first implemented on the IBM System/360* in 1966, it provided two mechanisms that allowed access to the environment: system commands and I-beams. Most APL users are familiar with system commands, since their use has survived and is widespread in current APL implementations. I-beams, on the other hand, are less familiar.

The use of the dyadic I-beam primitive was first introduced in APL\360 to allow execution of IBM System/360 instructions from within an APL program. It was considered an ad hoc facility for the use of system programmers, and was never formally accepted as a primitive or made part of the APL language. Nonetheless, I-beams were very useful and the facility was extended in later APL implementations. Monadic and dyadic definitions provided access to the underlying computing system. The definition of a dyadic I-beam required an integer left argument that specified the subfunction to be per-

formed, and a right argument and result that varied by subfunction. Monadic I-beams, whose right argument specified the subfunction, simply returned a subfunction dependent result.

In APL\360 and APLSV, the use of dyadic I-beams was restricted to privileged users and provided such functions as user and system control and access to memory. The monadic I-beams provided statistics on various aspects of the systems and access to certain key system variables such as time, date, and terminal type. All of the nonprivileged I-beams were replaced by system variables in later APL implementations (see Table 1).

Since the earliest implementations of APL, there have been requests from users for linguistic access to many of the functions provided by system commands. However, it was felt that the useful, usable, and rudimentary syntax of system commands did constitute a language—one that was incompatible with APL and had no constructive potential.[1] Locked functions were therefore provided in APL\360 to allow applications to perform such tasks as setting index origin, or the random seed. These locked functions contained I-beams that performed the actual work. Again, this provided an ad hoc solution to the problem. The long-term solution was implemented with the introduction of system functions and system variables in APLSV.

System functions and variables. In APLSV, two new types of objects, system functions and system variables, were introduced into the APL language. These objects, distinguished by names that start with the character □,[2] are defined in the implementation and are available in every clear workspace. In many senses, they are similar to primitives insofar as they provide specific predefined functions.

When system functions and variables were introduced into the APL language, they were introduced cautiously and only a few were provided. Unfortunately, their introduction was interpreted by some implementers as the long overdue solution to a serious problem—the problem that APL was limited, particularly in its access to system facilities. A number of APL implementers immediately reacted by introducing a large number of new system functions and variables. These functions and variables were introduced without much forethought, with little consistency in syntax or semantics, and with little compatibility between implementations. It was initially believed that system functions and variables were not part of the APL language, so implementers, perhaps installations, and maybe even individual users were free to invent as many as they pleased. System functions and variables, however, are very much a part of the APL language, as is demonstrated (in hindsight) by their inclusion in the APL standard. They now provide one of the more serious impediments to compatibility and portability.

Table 1 Nonprivileged monadic I-beams

I-beam	Description	Replaced By
19	Cumulative keying time	□AI
20	Time of day	□TS
21	Compute time since sign on	□AI
22	Free space in workspace	□WA
23	Number of users signed on	□UL
24	Elapsed time since sign on	□AI
25	Current date	□TS
26	First value in line counter	□LC
27	Line counter vector	□LC
28	Terminal type	□TT
29	User account number	□AI

Little thought was given to which functions should be provided as system functions, as primitives, or by means of other mechanisms. Very little guidance on this subject was provided to implementers. Functions such as **format** have been widely implemented both as primitives and system functions. Perhaps they are most appropriately neither; perhaps they should be defined functions. In the rush to provide commonly used, "omnipresent" functions with adequate performance, implementers have clearly gone overboard with system functions and variables. Fortunately, there have been no system operators introduced to date.

Component file systems. The need for file I/O was recognized as a key requirement in APL systems, before the introduction of system functions and variables. Component file systems were developed to fill this need and access to them was provided with locked functions that used the I-beam primitive. These locked functions were replaced with system functions soon after the introduction of those facilities. A typical component file system adds about 20 system functions to the language.

Component file systems provide facilities that allow APL arrays to be stored in and retrieved from external files. They are designed to be fast, straight-

forward, and simple to use in APL applications. They are not primarily designed to provide mechanisms that allow data interchange, via files, with non-APL systems. The file and record formats implemented in component file systems are typically complex and difficult to read or write from other high-level languages.

Shared variables and auxiliary processors. The introduction of shared variables with APLSV was motivated by the same need for file I/O facilities. Lathwell, Falkoff, and others who worked on this problem recognized that a primitive function or system function solution would eventually become unmanageable, particularly if a variety of access methods and file formats were to be supported:

> Most programming languages approach communication and storage problems by defining explicit communication primitives such as READ and WRITE to transfer information. These specialized primitives, used in conjunction with declarative statements and job control languages, result in programs which contain file-handling details irrelevant to the algorithm, and are strongly dependent on host operating systems and file structures. This approach was deemed inappropriate for APL because it conflicted with many of the principles that guide APL design; in particular, it conflicted with the requirement for machine-independent theoretical definitions of primitive functions.[3]
>
> . . . there is a high cost associated with the use of primitive functions for communication, as is the rule in most programming languages. This cost takes the form of complications in both syntax and semantics, and follows from the fact that in any language the arguments of a primitive function must be objects in the language. Thus, when functions like READ and WRITE operate on a variety of files, these files must necessarily be included in the language as additional constructs. The situation can become more and more complex, to the point where simple input and output statements are no longer adequate, and auxiliary statements, such as data declarations, must be introduced. These complications then make the language costly to implement, and costly to use.[4]

Further, Lathwell and others working on the problem realized that the requirement was not only for file I/O, but for other types of communication with components of the underlying computing facility. It was decided to implement a solution for the general communication problem, and to use that solution to implement file I/O facilities, among other things. The solution was shared variables, whose use had been originally postulated to describe channel architecture in the APL formal description of the IBM System/360.[5]

With APL2, variables may be shared between APL users on the same computer.

A shared variable differs from a normal APL variable insofar as it is "shared" or owned simultaneously by two "partners" or processes. Each partner can set or use the variable; its value at any given time reflects the last value set by either partner.

A control mechanism is provided to synchronize access to the variable, if such control is desired by the partners. If a shared variable is left uncontrolled, each partner is free to set or use the variable at will. With access control, however, protocols such as "master/slave" or "message passing" can be easily established.

Declaration, control, and management of shared variables is provided with a set of system functions. Variables can be shared between APL users or with other processes, referred to as "auxiliary processors," in the computing environment. Typically, auxiliary processors are programs written in a language other than APL that are designed specifically to share variables with APL applications and to provide specialized functions, such as file I/O, to those applications.

With APL2, variables may be shared between APL users on the same computer, between APL users and an auxiliary processor, or for that matter, between auxiliary processors. Auxiliary processors that exist in the APL user's address space are called "local" processors, and normally share variables only with that APL user. Auxiliary processors may also be implemented as multiservers that exist in separate address spaces and share variables simultaneously

with more than one APL user. Such auxiliary processors are called "global processors," and can provide facilities such as shared file support to a group of APL users.

Experimental facilities have been developed that allow variables to be shared between partners on separate computing facilities that are linked by telecommunication facilities.

Shared variables are handled by a component of the APL system called the "shared variable processor." This component is invoked when either partner attempts to set or use a shared variable. In most APL implementations, the shared variable processor uses an area of memory referred to as "shared memory" to temporarily hold the value of a shared variable until both partners are aware of it. Shared memory is also used to hold control and management information, such as identification, state, and access control for the shared variables and the partners sharing them.

Shared variables were designed to provide a general, asynchronous communication facility.

The initial implementation of shared variables in APLSV supported communication between APL users, and communication with auxiliary processors. One auxiliary processor, TSIO, was provided with the system, and it was expected that installations would write others as required. TSIO provided sequential and direct access to files maintained by the underlying operating system. It was particularly useful for exchanging files with applications written in other languages, but fell short in terms of function and usability when compared with the more special purpose component file systems.

There is no technical reason that a component file system should not be implemented with shared variables and an auxiliary processor. In fact, such implementations eventually emerged. At first, proponents of the component file system refused to consider the use of shared variables. In their defense, it should be pointed out that the use of shared variables was often difficult and complex before general arrays were introduced into the language. Auxiliary processors typically required paired variables and sometimes multiple modes of communication.

Further complicating the issue and polarizing those involved was the fact that many of the auxiliary processors that emerged were inelegant and inherently sequential in their communication protocol. Component file systems, on the other hand, typically presented a more elegant and usable interface.

Finally, it should be remembered that shared variables were designed to provide a general, asynchronous communication facility. It was originally envisaged that they would be used within cover functions to implement a specific communication protocol, or access method interface. Because these cover functions were not "omnipresent" or particularly good performers, however, and because most of the required communication involved simple synchronous protocols (e.g., READ, WRITE), the system function approach remained a more desirable alternative for many users.

When general arrays were introduced into the language, the use of shared variables and the implementation of auxiliary processors became considerably simpler. The command and data could be packaged together in a single WRITE request, and the return code and data could be packaged together for READ. Paired shared variables, with all of their associated complications, were no longer required.

Name association and external functions. Thus far, we have dealt mainly with issues involving file I/O. Since the emergence of APL there has been an additional requirement voiced by users for facilities that allow non-APL programs to be called from APL and to exchange data with APL. Over the years there have been a number of attempts to provide such facilities, typically with specialized auxiliary processors. While these auxiliary processors provided at least some of the needed function, their use never became widespread, probably for the following reasons:

- The auxiliary processors were difficult and cumbersome to use. Their use depended on shared variables for passage of control and data. Typi-

cally multiple variables had to be shared, and typically the interface was complex.
- The shared variable interface used was inherently asynchronous, while the primary requirement was for a synchronous interface to subroutines written in languages other than APL.
- Passing argument data was difficult. The shared variable processor sometimes imposed limits on the size of data that could be passed to a subroutine. Further, subroutines in other languages often required multiple heterogeneous arguments that were difficult to package and send across the shared variable interface.
- It was difficult to access routines that were not specifically designed to interface to APL. Existing libraries of subroutines required argument data types not supported by APL or specialized interface conventions.

General arrays presented a practical solution to some of these problems. They allow parameter passing on subroutine calls with a syntax amazingly similar to that commonly used in other languages, as shown in the following example.

APL:

```
A←10 20 30
B←'ABCDE'
C←1.2 1.3
PROCESS (A B C)
```

FORTRAN:

```
INTEGER*4 A(3)/10 20 30/
CHAR*5 B/'ABCDE'/
REAL*8 C(2)/1.2,1.3/
CALL PROCESS(A,B,C)
```

When this was recognized, it became clear that subroutines written in other languages could be treated syntactically as locked APL functions. To complete the design of this facility, "associated processors" were invented and the system function ⎕*NA* was introduced to declare a name to be external to APL.

⎕*NA* is used to declare the name of a variable, function, or operator to be external to APL and to be associated with a specified processor. When that name is subsequently encountered during execution of an APL expression, control is passed to the associated processor to perform the computation required to reference or specify the variable, or to execute the function or operator with the arguments and operands provided. On completion of this synchronous call to the associated processor, execution of the APL expression continues with any results returned.

The processing to be performed on an external name when control is passed to its associated processor is not defined in the APL language. An APL system may provide many associated processors to deliver different sorts of function to the APL users. When this facility was initially introduced in APL2 Version 1, Release 2, two associated processors were supplied to provide support for calls to routines written in FORTRAN, assembler, and REXX. Since that time, users have used these processors to call a wide variety of routines and languages including PL/I, COBOL, C, and Pascal.

The problem of argument coercion to the data types expected by the external routines in languages like FORTRAN was solved by providing facilities in the associated processor to allow descriptive information to be associated with any of the called routines. This information, among other things, provides descriptions of the expected arguments and their data types for an external function. When the function is called, it is used to determine if the expected arguments have been provided, and if the data types of those arguments need to be transformed to data types expected by the external function. A similar process is used to transform results from the external function to data types acceptable to APL.

One of the real advantages of this solution to the requirement for calls to non-APL routines is that these external routines look just like APL locked functions. Thus it is possible to write an application entirely in APL and then replace portions of it with routines written in other languages; or it is possible to design a heterogeneous application without doing damage to the syntax of the APL portions of that application.

In APL2 Version 1, Release 3, the facilities supporting external functions were extended to allow external functions called from APL to issue calls back to APL. Using these facilities, non-APL routines can request execution of APL functions or operators, or can reference or specify APL variables. This extension could be particularly useful for external operators whose operands might be APL functions, or for an APL compiler that might choose to compile parts of an application but use APL primitives for other parts.

Recently, an enhancement to APL2 Release 3 was made to allow non-APL application programs to invoke APL and issue calls to it. Using these facilities, applications written in a wide variety of languages can conveniently and simply execute APL functions, passing arguments to them and receiving results from them. Using the same facilities, the non-APL application can also reference or specify APL variables, or pass control to the APL interactive environment.

Namespaces represent an important advance in APL systems.

APL namespaces. When external functions and associated processors were designed, the interface was structured such that calls to routines written in APL could be accommodated. In particular, ambivalent functions and operators were not excluded in the interface syntax.

After considerable discussion and experimentation, it was decided to use this facility to address the problems of name scope isolation and shared code for APL applications.[6]

With an extended interface provided in APL2 Release 3, it is possible to declare an APL variable, function, or operator to be external to the workspace and to exist in another "namespace." A namespace differs from an APL workspace in two ways. First, it is formatted to allow it to be handled by the operating system facilities used to load programs, rather than in the normal format of a saved workspace. Second, it is accessed in a read-only mode; the results of computations are never actually stored in a namespace, but rather in the user's active workspace from which the namespace was accessed.

Like the active workspace, each namespace defines a name scope. A name scope is simply a set of names of variables, functions, and operators and the values and definitions associated with them. Users are able to declare names to exist in a namespace, in much the same way that external function names are declared with *⎕NA*. When the name of an external APL function, operator, or variable is encountered during the execution of an APL expression, the system locates the namespace in which it exists and switches to the name scope of that namespace in order to process that name.

For an external APL function, this means that arguments to the function are provided from the caller's name scope, but names referred to in the body of the function come from the namespace's name scope. For an external APL variable, it means that the value comes from the namespace name scope when the variable is referenced, and is set in the name scope of the namespace when the variable is specified.

Since namespaces are accessed on a read-only basis, they may be shared between users. New or modified values or function definitions in a namespace name scope are actually saved in the user's active workspace. Thus, if more than one user accesses the same namespace, the system behaves as if each has its own private instance of it. Further, the state of the namespace, if modified as a result of execution, is maintained and can be saved and reloaded along with the workspace with which it is associated.

Namespaces represent an important advance in APL systems:

- They provide a simple, convenient, and powerful way to segment applications and to deal with the problems of "name pollution" common in large applications.
- They allow dynamic access to segments of an application without *)LOAD* or *)COPY* commands.
- They provide a mechanism where application programs can be shared by multiple simultaneous users; this is particularly important for large popular APL application packages.

Comparison of facilities

As previously described, there are three major facilities provided in the APL language that allow access to things outside the APL workspace: system functions and variables, shared variables, and name association.

Had all three of these facilities been proposed for incorporation into the APL language at the same

time, all three probably would have been accepted. Clearly, there are advantages and useful applications for each of the facilities. It should also be clear that there is a substantial amount of overlap in the applications for which each facility has been used. Many applications could be implemented with any one of the facilities, and the specific choice that was made in many cases reflected the state of APL implementations at the time, rather than any particular reason that one facility was better for an application than another.

System functions and variables offer the advantage that they are "omnipresent," and create no name conflicts with application-defined names. A unique function or variable, however, is required for each distinct operation. Unless restrictions are placed on implementers, this will inevitably lead to a large and unmanageable number of system functions and variables, and conflicting names between implementations. The APL standard defines about 20 system functions and variables; APL2 defines 41; another popular implementation defines over 120.

Some system functions and variables are clearly part of the language and are required for execution of most applications. ⎕IO, ⎕CT, and ⎕NC are certainly in this class. Further, it is appropriate that they be implemented as system functions and variables rather than primitives, because they have to do with the implementation of APL as a programming language, rather than as a machine-independent language. Other functions like ⎕SVO or ⎕NA must be implemented as system functions if they are to provide access to facilities that in turn provide extra-linguistic function.

It is not clear, however, that system functions and variables like ⎕DL, ⎕ARBOUT, ⎕AI, and ⎕UL should be part of the language. None of these is required for proper operation of the primitive functions and each could easily be implemented as an external function or with shared variables.

There are no explicit rules or guidelines to tell implementers whether a facility should be implemented as a system function, a primitive, or an external function. There is some consensus that primitive functions should deal only with abstract objects (arrays of numbers and characters), while management of the APL environment or interface to things outside the APL environment should be provided with nonprimitive functions. All of the system functions defined in the APL standard or APL2 have to do with APL as a computer programming language, and thus are appropriate nonprimitives. There are, however, a number of primitive functions like ⍋, ⍒, ?, and ⌹ which might better be implemented as something other than primitives.

The distinction between the shared variable and name association facilities is a little clearer. Shared variables implement a general-purpose, asynchronous communication facility between cooperating but independent processes. Name association, on the other hand, allows the processing associated with function call and variable reference or specification to be handled in a synchronous manner by an external processor and in a name scope other than the user's active workspace.

There is some consensus that primitive functions should deal only with abstract objects.

Because system functions and shared variables predated the implementation of name association, these earlier facilities were sometimes used to implement function that is more appropriately handled by name association. File I/O is a good example. There is a need for access to many different file subsystems from APL, which often require the use of different syntax and arguments and whose use may be desirable in one application but not in another. Typically, the access to file subsystems is most conveniently implemented with synchronous subfunction calls, rather than with the more complicated shared variable interface. Because of the diverse requirements for functions to handle these interfaces and because of the number of functions required for full support of an access method, it makes most sense to implement these functions as external functions rather than system functions. One final advantage of the external function approach is that it is possible in some cases to change access methods by merely changing the name association of the external functions.

Another class of functions that are more appropriately provided as external functions include ?, ⌹,

dyadic ⍕, and □FMT. Each of these functions implements one of a set of acceptable solutions. For example, ? generates random numbers with a flat distribution. While this is acceptable in many applications, there are certainly lots of other applications where other distributions would be more appropriate. Where functions exhibit this characteristic, they should be provided as defined or external functions rather than primitives or system functions.

Choice of the correct facility. From the foregoing, it should be clear that the choice of a "correct" facility for the implementation of a specific function is not simple. There are no clear-cut guidelines, and many new proposals fall into grey areas. Nonetheless, there are some principles that should be kept in mind when choosing a facility to implement specific function:

- APL is designed to be an abstract language whose definition is machine-independent and need not be associated with a computer system in any way. Primitives in the language should adhere to these principles.
- Primitives in the language should be useful across a wide variety of applications and a wide variety of users. Further, they should be general and usable in conjunction with other primitives to provide rich function.
- Function should not be implemented as primitive where only one of a set of commonly acceptable solutions is implemented. Random number generation is an example of such a function. It is useful only if the particular mathematical algorithm used is appropriate to the user's problem.
- System functions and variables are part of the language. Users should be able to depend on their availability across implementations. Use of a system function or variable should not inhibit the portability of an APL application.
- There is no such thing as a primitive variable. Thus, variables such as □IO or □CT, which are implicit arguments to primitive functions, are appropriately implemented as system variables.
- Functions that are needed to declare the machine-dependent characteristics of an APL object (such as "shared variable" or "external function") are appropriately implemented as system functions.
- Functions required to manage the contents of a workspace, such as □NC, □NL, □CR, and □FX, are appropriately implemented as system functions. Care should be exercised in this area, however, since other commonly accepted system functions such as □TF can be easily defined based upon ⍕, □CR, and □FX. Redundant function should be avoided.
- The availability of external functions and variables makes it possible to implement a great deal of commonly used function with acceptable performance characteristics. In a large number of cases, external functions and variables are a more appropriate implementation vehicle than system functions and variables.
- External functions use a synchronous interface to facilities outside APL that can be thought of as a subroutine call. Shared variables, on the other hand, provide an asynchronous communication channel and are more appropriately used where this asynchronous characteristic is important.

Improvements and extensions

Given the opportunity to do it all again, there are certainly some things that would be done differently. In a perfect world, implementers would be more clairvoyant and would easily choose between primitives, system functions, external functions, and shared variables. Unfortunately, given the broad base of existing users and their investment in APL application code, it will be difficult to make any radical changes in the short term. Existing facilities will have to continue to be supported, probably for a considerable length of time. We can hope, however, that as new function is implemented, appropriate facilities will be used, and that the benefits inherent in the use of that new function will quickly attract users.

With regard to the facilities themselves, however, a number of improvements and extensions can be envisaged:

- While the use of system functions and variables to implement new function should be avoided in many cases, the usability of system variables could be improved with a simple extension. If pass-through localization[7] was provided for system variables, certain operations, which are cumbersome now, could be made much simpler. For example, with pass-through localization a function could easily capture its caller's □IO before setting its own:

```
    ∇Z←L F R;⎕IO;IO
[1]  IO←⎕IO ⍝ GET CALLER'S ⎕IO
[2]  ⎕IO←0  ⍝ BUT USE ⎕IO←0
     .....
```

- It is sometimes possible to make simple changes to auxiliary processors that result in substantial performance or usability improvements. For example, APL2's AP 111 has been extended recently to support matrix output. It could also easily be extended to support matrix input.

- Variables in APL namespaces are currently copied into the user's workspace before they are used. It was just simpler to implement the system that way. An obvious extension would allow external variables to be used without first having to make a copy of them. With such an extension, namespaces could be used as data spaces housing large, shared, in-memory tables of data.

- Shared variable processor facilities could be extended to allow communication between physical machines. Such an extension might be particularly useful between APL applications running in a client/server relationship, for example, between workstation and host-based applications.

- Similarly, associated processors could be developed to generate remote procedure calls to cause external functions to execute on a different physical machine. Again, such an extension would be particularly useful to a workstation APL implementation where the power and facilities of a host machine might be highly attractive. Such an extension would allow true distributed processing without any change to the language or to many existing applications.

- The introduction of external functions and associated processors into APL represents an important advance, allowing hybrid applications to be constructed from a variety of tools or languages. The facilities provided with APL2 are nonetheless relatively rudimentary at the present time and could be extended and simplified to make the construction, testing, and maintenance of such hybrid applications considerably simpler.

- As described in this paper, facilities to perform input/output (e.g., file I/O, screen I/O, etc.) have been implemented in a variety of ways including locked functions, system functions, shared variables, and external functions. All of these implementations introduce a degree of complexity to the APL user who simply wants to treat data as data, irrespective of the source or destination. The introduction of large workspaces in APL2 demonstrated that when all data used by an application could be maintained in APL variables in the workspace, the complexity of the application was often reduced substantially. The technology provided with associated processors, if extended in a few areas, could provide a mechanism that would allow data on files, or for that matter, data on the user's screen to be treated by the application as if the data were resident in variables in the user's workspace. Indeed, limited forms of this approach have been implemented in some systems with shared variables or system variables used to access external data. The use of external variables and associated processors offers an opportunity for generality and power not afforded by earlier approaches.

These examples of improvements and extensions range from suggestions that would make the facilities in today's APL implementations more usable and more valuable, to extensions that open up new opportunities for APL applications and for the exploitation of system facilities from an APL environment.

Conclusion

APL was originally conceived as a mathematical notation used to express ideas and algorithms. When it was later found to be a useful computer programming language, it became evident that its domain had to be expanded to provide connectivity to systems and facilities outside the APL workspace.

The mechanisms that provide connectivity between APL and other facilities in the computing environment have evolved over more than 20 years. There is no evidence to suggest that this evolution is complete. In fact, it seems to have been accelerating recently. In the first 20 years, we made many mistakes by rushing to use existing interfaces to solve all problems, often without a good understanding of the interfaces and without attempting to determine whether completely new types of interfaces need to be developed. The unfortunate part of this story is that users have made substantial investments in application code that is often difficult and costly to migrate to new and better facilities as they emerge.

The wise APL application developer develops an application as a set of building blocks that can be replaced as better technology becomes available.

* Trademark or registered trademark of International Business Machines Corporation.

Cited references and notes

1. A. D. Falkoff and K. E. Iverson, "The Design of APL," *IBM Journal of Research and Development* **17**, No. 4, 324–334 (1973).
2. Formally, the names of system functions and variables may begin with either □ or ⍞; however, to date no ⍞ names, other than ⍞ itself, have been introduced.
3. R. H. Lathwell, "System Formulation and APL Shared Variables," *IBM Journal of Research and Development* **17**, No. 4, 353–359 (1973).
4. A. D. Falkoff, *Some Implications of Shared Variables*, Technical Report 02.688, IBM San Jose, CA (June 1975).
5. A. D. Falkoff, K. E. Iverson, E. H. Sussenguth, "A Formal Description of System/360," *IBM Systems Journal* **3**, Nos. 2 and 3, 198–261 (1964).
6. It should be noted that this choice is implemented by a particular associated processor provided with APL2. Other associated processors could be implemented to offer other choices to the user.
7. With pass-through localization, a local variable retains its global value until specified.

Accepted for publication July 24, 1991.

Michael T. Wheatley *IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141.*

Mr. Wheatley is currently a Senior Technical Staff Member in the language products development organization in the IBM Santa Teresa Laboratory. He has been involved with APL marketing, support, and development within IBM for over 20 years. From 1979 to 1989 he worked with James Brown on the design and implementation of APL2. As part of that effort, Mr. Wheatley led the design and implementation teams for the shared variable processor, auxiliary processor, associated processor, external function, and namespace components of APL2. Mr. Wheatley graduated with a B.S. in mathematics from the University of Montreal in 1966. He holds two patents, one patent on file, and one published invention disclosure, all of which are APL-related. He is a recipient of an IBM Outstanding Innovation Award for his work in the design and implementation of APL2 namespaces. Mr. Wheatley is currently the cross language architect in the Santa Teresa Laboratory with lead technical responsibility for IBM language products.

Reprint Order No. G321-5445.

Storage management in IBM APL systems

by R. Trimble

APL systems have traditionally used specialized storage management schemes that avoid storage fragmentation by "garbage collection," moving live data as needed to collect unused storage into a single area. This was very effective on systems with a small amount of real storage addressed directly. It has become less effective on today's systems with virtual addressing and large amounts of virtual storage. Both traditional schemes of storage management and a recently implemented replacement for them are described. The focus is on implementations for IBM mainframe hardware.

Programs written in compiled languages typically use static definitions of working storage. Much of the time the language syntax requires that variables be declared as a particular type, structure, and often a particular size. This allows the compiler to generate very specific code for accessing the variables.

In contrast, interpretive programs typically provide much less data declaration information, and declarations are often implicit in the data usage. A number of interpretive languages allow a single variable to take on varying definitions at different times during program execution. APL, in fact, has no data declaration constructs at all for objects that exist within the active workspace. An object may change during execution from Boolean to real to complex, from simple scalar to four-dimensional array to nested structure, and from numeric to character to defined function.

Depending on the point of view, this dynamic characteristic of data has been described as introducing anarchy into the language, forcing heavy execution time overhead, or permitting powerful and elegant algorithms that are independent of data structure and format. Less frequently analyzed is the impact on storage management strategies, and the secondary impact of those strategies on total system performance. This paper discusses the storage management schemes that are used for APL running on IBM mainframe processors.

APL data organization

By necessity, APL objects must be completely self-describing, and it is impractical to assign them fixed locations or sizes. This leads immediately to a level of indirection in locating named objects accessed by programs or users. Ultimately the locator technique must provide for a symbol table lookup, since new references to objects can be introduced interactively at any time. In practice, though, a symbol table search incurs too much overhead on every reference, so a pointer table with statically assigned slots is used, each slot pointing to the current location of the associated data. Programs needing to access a data object can retain a table index for that object instead of its actual address.

Traditional APL implementations are contrasted here with systems like LISP that have large numbers of internal connections among relatively small stored objects. Some APL systems, such as VS APL,[1] did use internal synonym chains to avoid making copies of objects, but in general APL systems have

handled large array objects that were mostly *externalized*, or named. LISP and some other languages typically use direct internal pointers from one object to another, and this is the only reasonable approach when storage cell sizes are very small. A full pointer table for LISP could easily use up a quarter or more of all available space in the system, and management of space within it could become a severe problem.

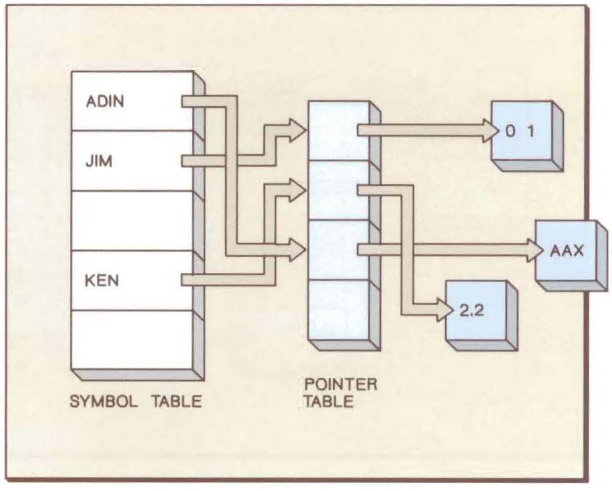
APL2 has introduced nested arrays into the language, and this has significantly increased the number of internal connections, but the array orientation remains. For this reason, and to avoid the decision overhead of handling a mixture of direct and indirect pointers, APL2 follows traditional APL usage of making all pointers indirect.

There are two major ways in which pointer tables have been implemented by APL systems. Figure 1 shows separate symbol tables and pointer tables. This approach permits the symbol table to be structured for binary or tree searches, and to be reorganized or expanded as needed.

Figure 2 shows a combined symbol and pointer table. The names of objects are stored as if they were objects themselves (though some systems store short names directly within the table). To locate a symbol by name, the system must follow the name pointer from each row of the symbol table. The combined table requires less storage, but is not amenable to table reorganization, since an unknown number of indices into it exist throughout storage. Typically a hashing scheme is used to locate names within the table, but this precludes dynamic expansion of the table. Table expansion would be possible only if sequential searches were done (which are very costly in time) or if an index were maintained (still significantly more costly than hashing). For these reasons, systems employing the combined table normally have a fixed symbol table size, or a size that can only be set when an APL workspace is first created.

The combined table was used in earlier APL systems, including IBM's APL\360, APLSV, and VS APL. APL2, IBM's current offering for the IBM System/370* and System/390*, uses separate symbol and pointer tables, in large part because of nested array extensions, but also partly because it was designed for large applications with more objects, making symbol table expansion much more important.

Figure 1 Separate tables for locating objects

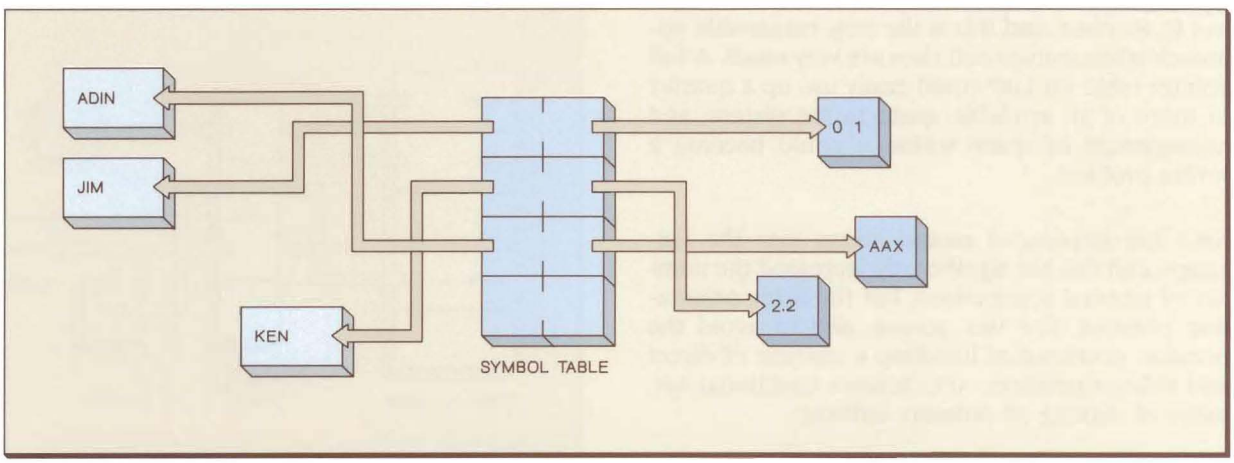


Whichever structure is used by an implementation, the primary pointer table contains addresses of all other objects in the APL workspace, and that is the only place (with occasional exceptions) such addresses are kept. Since interpreter routines always maintain a direct pointer to the pointer table, there is very little extra cost in converting a table index to the address of the corresponding object. Most importantly for storage management, it is also easy to move an object from one place to another, since only a single pointer to it needs to be updated. One other rule is enforced to make this possible—any pointers to locations within an object are always stored as offsets, not addresses.

Traditional APL storage management

Another attribute of APL objects is that many of them are very transient. APL programs often use simple names like *X* for variables that contain many different kinds of data during the execution of a single defined function. Since the storage requirements for these various usages may gyrate wildly, the system actually creates a new object each time a value is assigned to the variable, and discards the object that previously represented the variable (thus their transient nature). Also, because APL is an array processing language, intermediate results are arbitrarily large and it is not practical, in general, to use predefined temporary areas to hold those results. Thus each processing step within an APL statement produces a new object as its result.

Figure 2 Combined table for locating objects



Because of these characteristics, storage allocation and release are critical paths in the performance of APL systems. Operating system path lengths for allocating and freeing storage are typically hundreds or thousands of instructions. If APL were to use those services for each object allocation, they could easily use up 90 percent or more of the application execution time. Thus APL, along with a number of other languages, was compelled to provide its own storage management function within an area (which APL calls an active workspace), obtained from the operating system.

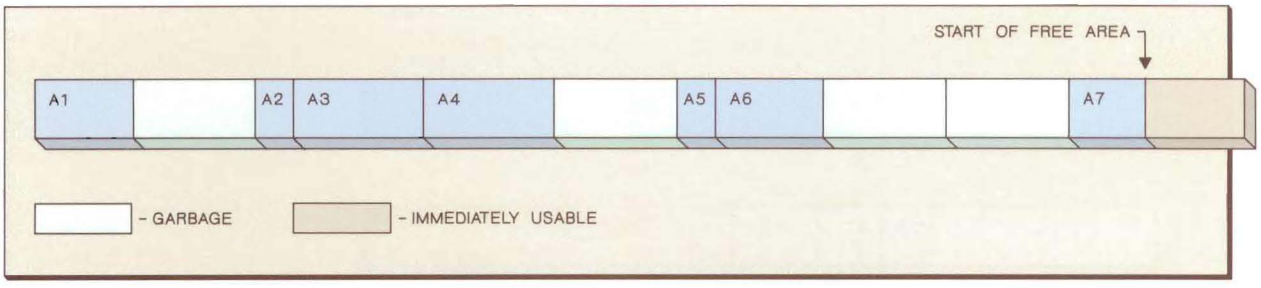
The traditional APL storage management technique is very simple, but extremely fast in processing time. The APL system adds a standard prefix to all objects. The size and format of the prefix have varied among APL implementations, but the prefix has included at least a flag (typically the first bit) that indicates whether the area is currently in use or is *garbage*, i.e., data no longer needed, and a field containing the length of the area.

A pointer is maintained to the beginning of a free area where it is known that no storage is currently allocated. When an allocation request is made, the storage is allocated at the beginning of the free area, and the free pointer is stepped beyond the new allocation. When an area is freed, its *garbage* flag is set. (Often the end of the freed area is checked against the free pointer; if they match, the free pointer is backed up, but this is not a necessary part of the algorithm.) Figure 3 shows a simple example of what a workspace might look like after a few such storage operations.

Eventually, of course, the free pointer will approach the end of the free area, and a storage request will be made that cannot be satisfied. This triggers *garbage collection*, which has a number of meanings in computing literature:

1. Garbage collection sometimes refers to the process of determining which parts of storage can be reused, perhaps by following all valid storage links. APL, as was already indicated, maintains a garbage bit in each block of storage. It also maintains (either in the storage block or the pointer table) a *use count* (i.e., storage is in use) field for each active block. The garbage bit is turned on when the use count goes to zero, so there is no ambiguity about which blocks of storage can be reused.
2. When a garbage bit is available, the system normally, at some time, scans storage looking for blocks it can reclaim. Often part of that scan involves coalescing adjacent garbage blocks. APL garbage collection performs this process.
3. Blocks identified as containing garbage may be chained together for later reuse. This has not typically been done by APL systems, because it does nothing to relieve fragmentation and es-

Figure 3 Sample of workspace with garbage, where A1-A7 represent allocation requests that have been satisfied



sentially leads to the same sorts of operating system storage management schemes and path lengths that APL has tried to avoid.

4. "Live" blocks (those containing data that are currently in use) may be moved, resulting in adjacent areas of garbage that may be collected into larger garbage blocks. Since APL maintains a complete indirect pointer list, it is relatively simple to move live entries. (APL systems normally maintain a pointer list index in the live entries, which makes it trivial to locate the one pointer which must be updated.) So for APL, garbage collection is the process of returning all of the garbage areas to the block of free storage.
5. There are several possible algorithms. APL systems have almost universally used a "shifting" rule that keeps the live storage blocks in their previous order. The advantage of this is that over time the more static objects in the workspace will migrate to the low-address end, and will be unaffected by later garbage collections. (Typically a "lowest garbage" pointer is maintained so that the system can skip over the static part of the workspace.) The disadvantage is that in the short term very large amounts of storage may need to be moved to make small but previously long-lived blocks reusable.

Some APL systems have used *predictive garbage collection* techniques that do the storage compaction as soon as a certain amount of garbage has accumulated. This approach can eliminate long pauses for garbage collection at unexpected times, but typically also increases the number of times that an object will be moved before it reaches its final resting point (or is deleted). Thus the net effect of such schemes is to increase the total amount of storage movement in the system, and so increase the CPU time used in processing an application. The approach can be useful despite this characteristic, both because it does yield a more predictable response time, and because it can reduce the application working set. More will be said later about that aspect.

One other enhancement used by VS APL[1] was to "ping-pong" allocations between both ends of the free area. It did this by maintaining floating pointers to both the beginning and the end of the free area, and by alternating their usage. The usefulness of this becomes apparent when we consider what happens while processing a series of primitive functions in an APL statement. For example:

```
A←1 3 2.5
A←2+3×⌊A
```

Figure 4 illustrates the sequence of allocations with a normal single-ended workspace system. Note that each primitive operation must obtain space for its result and calculate it before the space for the previous temporary result can be released.

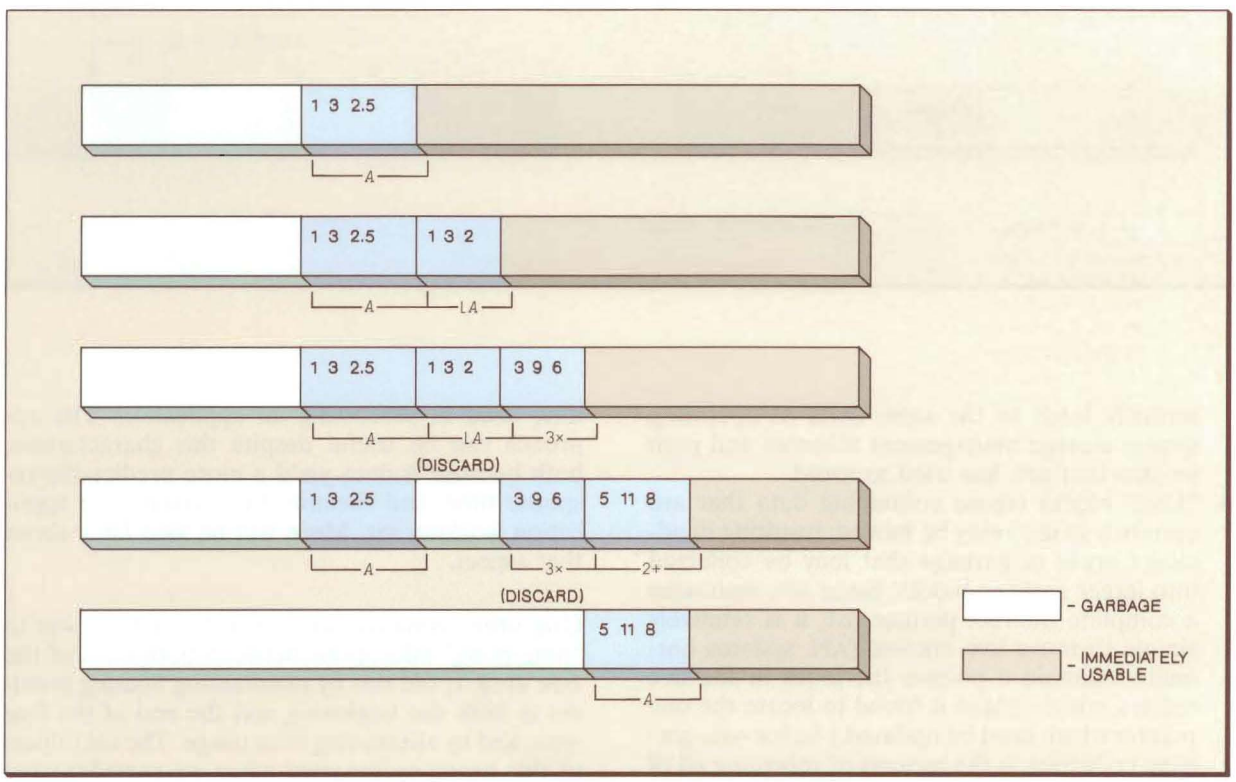
Figure 5 shows the corresponding sequence of allocations with a two-ended (ping-pong) workspace.

Because of the ping-ponged allocations, temporary blocks can often be returned immediately to the free area, and embedded garbage builds up more slowly.

The costs of garbage collection

The first APL implementations ran on systems without paging facilities and used 32K-byte work-

Figure 4 Processing with a one-ended workspace



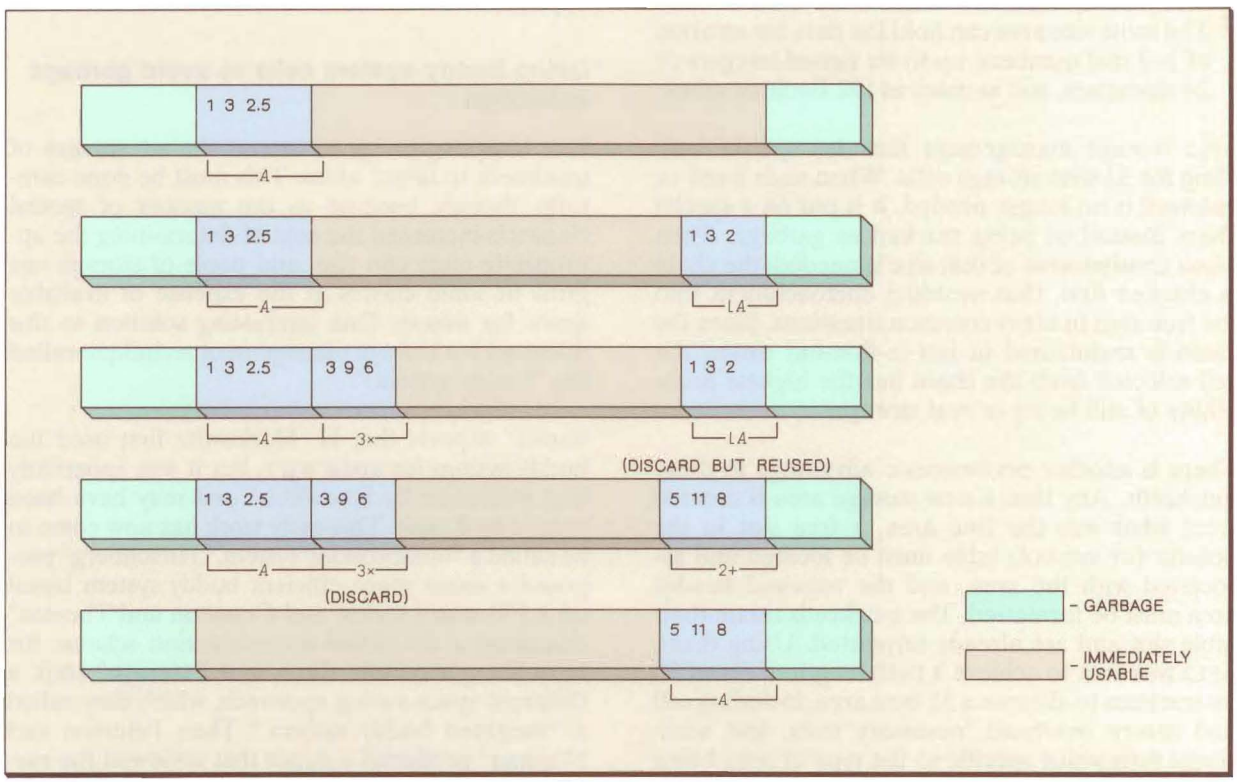
spaces. A typical garbage collection would move 4K bytes of data or less, and might use the time equivalent of less than 1000 instructions.

Today's APL products run on systems that are often capable of supporting workspaces up to a gigabyte or two in size, all in pageable virtual storage. Although many users limit themselves (or are limited by their installations) to 10-megabyte workspaces or less, a significant number are routinely using 50–100 megabytes or more. In typical cases only a small part of these larger workspaces is used for static data and functions. The extra space has made it possible to manipulate multiple megabyte arrays and use algorithms with very large intermediate results. But this in turn means that garbage collections often involve moving many megabytes of data.

A typical garbage collection for a 20-megabyte workspace might move 2–4 megabytes of data, requiring execution time equivalent to executing on the order of 100 000 instructions. But this is only the beginning of the problem. In the process of locating and moving the data, the APL system will probably touch 75 to 80 percent of the pages in the workspace, or around 400 pages for the 20-megabyte example. On typically loaded multiuser systems a significant number of these will be paged out, resulting in long delays to retrieve them, one after another. These delays can easily add up to execution pauses of 5 to 10 seconds, which is intolerable in an interactive system. These sudden paging loads can also trigger periods of saturation for the paging devices, and thus lead to execution pauses for other interactive users on the system.

Finally, periodic usage spikes of real storage caused by garbage collection mislead system resource management programs, causing them to overestimate future APL real storage requirements and frequently to move APL users to a lower priority service class for most of their processing.

Figure 5 Processing with a two-ended workspace



The scenario just described at 20 megabytes becomes much (more than ten times) worse at 200 megabytes. The system has a limited amount of real storage, and only a fraction of that can be dedicated to a single user. It is a rare system today that will allow one user to control as much as 150 megabytes of real storage at a time. Note that APL garbage collection actually involves two pointers "floating up" through the workspace, one for where blocks are being moved to and the other (many megabytes ahead of the first in such a huge workspace) for where blocks are being moved from. When the distance between those pointers exceeds about half the real storage of the available user storage in the system, the pages will begin to be paged out and back in again between the time they are used. This can triple or quadruple the paging load described above. Execution pauses of many minutes have been reported under these circumstances.

As was indicated earlier, predictive garbage collection, taking action when a threshold is reached on uncollected garbage, actually increases the total amount of storage movement and thus the processor time required to run a given application. Despite this, it can be useful, because the paging usage spikes are significantly reduced, and the "moved to" and "moved from" pointers are much closer together during a garbage collection.

Using quickcells to minimize garbage collection

APL2 uses one very successful strategy to reduce the number of garbage collections. Although object allocations come in many and varying sizes, it was noted that a large number of them are quite small. This is particularly true for APL2, which actually uses two or more separate storage areas for most nonscalar data objects. One of the areas contains the data objects themselves and the other contains the description of the data. (Nested arrays include a number of descriptor areas and data areas.)

- The standard descriptor blocks for vectors, matrices, and three-dimensional arrays can all be fit into a 32-byte area.
- The same size area can hold the data for an array of 1–3 real numbers, up to six signed integers or 24 characters, and as many as 192 Boolean values.

APL2 storage management includes special handling for 32-byte storage cells. When such a cell or *quickcell* is no longer needed, it is put on a special chain instead of being marked as garbage. Then when another area of that size is needed, the chain is checked first, thus avoiding encroachment into the free area in many common situations. Since the chain is maintained in last-in-first-out order, the cell selected from the chain has the highest probability of still being in real storage.

There is another performance advantage to these quickcells. Any time a new storage area is created from what was the free area, a free slot in the pointer (or symbol) table must be located and associated with the area, and the required header area must be formatted. The quickcells retain their table slot and are already formatted. Using them, APL2 was able to achieve a path length of about 30 instructions to allocate a 32-byte area, including call and return overhead, necessary tests, and additional formatting specific to the type of area being obtained.

APL2 also provides a separate quickcell pool for each type of scalar data (a single unstructured character or number). There are six of these pools, covering everything from standard characters to complex numbers. Four of those six (characters, extended characters, short integers, and signed integers) need only a 16-byte area, so the system splits a quickcell to form two "short scalars." Allocation path lengths for all of the scalar quickcells are a trivial 11 instructions because special entry points are used, only one test (for empty pool) is needed, and the cell is already completely formatted except for the actual value.

Of course it is possible for APL applications to use a very large number of such areas for a short time, leaving huge pools of quickcells behind. The normal garbage collection algorithm would not detect those, but APL2 provides a special quickcell cleanup routine that does release the table slots and mark the cells as garbage. This is usually performed if a standard garbage collection is not able to free up enough space. Until such time as this happens, though, large quickcell pools can have the effect of increasing the number of real pages required by the application.

Using buddy-system cells to avoid garbage collection

It is tempting to try to extend the advantages of quickcells to larger areas. This must be done carefully, though, because as the number of special classes is increased the cost of determining the appropriate class can rise, and pools of storage can grow in some classes at the expense of available space for others. One interesting solution to this dilemma is a storage management technique called the "buddy system."

Knuth[2] reports that H. Markowitz first used the buddy system for SIMSCRIPT, but it was apparently first published by Knowlton[3] and may have been named by Knuth. This early work has now come to be called a "binary buddy system." Hirschberg[4] proposed a more space-efficient buddy system based on a Fibonacci series, and Cranston and Thomas[5] described a simplified recombination scheme for Hirschberg's system. Shen and Peterson[6] took a different space-saving approach, which they called a "weighted buddy system." Then Peterson and Norman[7] produced a paper that reviewed the various buddy schemes and concluded that either the original binary system or the improved Fibonacci system was preferable. Bozman et al.[8] found buddy systems in general very fast but inferior to subpool-based systems for their purposes with IBM's VM/SP product. This may have been because the binary system described below required an additional double word in each allocation. As will be shown, no extra storage is needed for APL.

Unfortunately for APL2, the improved Fibonacci system depends on availability of three status bits in the storage block, which would require a major restructuring and reinterpretation of flags throughout the interpreter. So the following information focuses on the original binary system. It should be noted that Page and Hagins[9] have more recently defined an improved weighted buddy system, but we have not analyzed that for applicability to APL.

The binary buddy system works by allocating all storage in sizes which are a power of two. Free area chains are maintained for each storage size. If no storage is available on a particular chain, an area can be taken from the next larger size and split to form

two areas, one of which will be put on the chain and the other used to satisfy the current request. (This splitting is, of course, a recursive process since the next larger chain could also be empty.)

Consider what would happen if a request were made for 80 bytes of storage and the system currently had no free blocks smaller than 4K bytes. The initial request would be rounded up to the next power of 2 (128 in this case) and then recursive splitting would be used to satisfy it. An implementation can choose which of the two "buddies" created by splitting an area is to be used immediately, and which is to be placed on the chain. For this example we assume that the buddy at the lower address is placed on the chain. Figure 6 is a pictorial representation of the way the 4K-byte storage area would be divided up at the end of the request.

First, note that any request for a small amount of storage when pools are empty will not only get that storage but will prime all pools up to the next one that was not empty. Because of this behavior, pools tend to be nonempty much of the time, and a majority of storage requests can be satisfied without having to split larger cells.

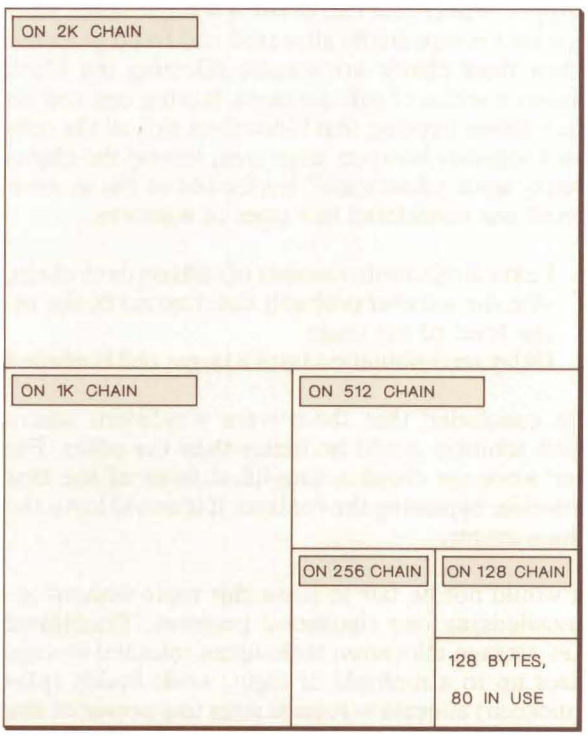
A second less obvious observation is critical to the behavior of the buddy system when storage is returned. If the original 4K area illustrated in Figure 6 began on a 4K-byte boundary, then the 2K buddies will each be on a 2K-byte boundary, 1Ks on a 1K boundary, and so forth, no matter how many times the area is split. In general any buddy system cell must be on a boundary that is a multiple of its size, and that requirement will be met automatically so long as it was met by the original areas. A different way of stating this is that for any buddy cell of size 2^n, the low order n bits of its binary address will be zero.

Now consider what happens when a cell of size 2^{n+1} is split. The first (low address) cell created will have $n + 1$ low order zero bits in its address, while its buddy will have the same address except for a 1 in the first of those $n + 1$ low order bits. Splitting a 1K (2^{9+1}) cell, for example, yields Boolean addresses of

```
xxxx xxxx xxxx xxxx xxxx xx00 0000 0000
xxxx xxxx xxxx xxxx xxxx xx10 0000 0000
```

But that same bit position is the location of the sole 1 in the binary representation of the length (2^n) of the new buddy cells. This leads to the remarkably

Figure 6 Dividing a 4K storage area to satisfy an 80-byte request



useful conclusion that given any buddy cell, performing an *exclusive-OR* operation of its address with its length will yield the address of its buddy.

The exclusive-OR technique makes it feasible to coalesce cells without an unreasonable amount of processing if two pieces of information are available with each cell:

- A flag that indicates whether the area is currently in use
- A field containing the length of the area

These are the same pieces of information that APL storage systems have always maintained. The buddy can be located by using the exclusive-OR operation. Once located, the two areas can be coalesced if the buddy is not in use and if it has not been further subdivided, i.e., if its length has not been reduced.

Knowlton's original paper expressed one concern that most later researchers seem to have ignored.

He felt that it might be better not to coalesce buddies in all cases where that was possible. Based on some modeling we indeed found what we called a "zipper" effect that can occur if a single small storage area is repeatedly allocated and freed at a time when most chains are empty. (Getting the block causes a series of cell divisions, leaving one cell on each chain. Freeing that block then zips all the cells back together into one large area, leaving the chains empty again.) Kaufman[10] has looked at this in some detail and considered two types of solutions:

1. Leave a minimum number of cells on each chain, with the number probably determined by the usage level of the chain
2. Delay recombination until a larger cell is needed

He concluded that there were conditions where each solution would be better than the other. For our work we chose a simplified form of the first solution, bypassing the coalesce if it would leave the chain empty.

It would not be fair to leave this topic without acknowledging one significant problem. Traditional APL storage allocation techniques rounded storage sizes up to a multiple of eight, while buddy (plus quickcell) allocation rounds sizes to a power of two with a minimum of 16 bytes. This has been referred to in the literature as *internal fragmentation*, and can result in an effective virtual storage utilization of only about 75 percent.

That number can be intuitively understood by observing that all of the storage areas allocated to a given buddy cell size are at least 50 percent as large as that cell, and at most 100 percent of the cell size. Assuming a linear distribution of sizes, the size of the required storage would average 75 percent of the cell size. In practice, size distributions are skewed with more allocations of the smaller sizes, so that the typical utilization should be somewhat less than 75 percent. Compensating for this is the fact that more than half of the allocations are for either scalar quickcells or array descriptors in quickcells. And it happens that those always use at least 75 percent of the cell size.

Buddy system researchers have also explored *external fragmentation*, which occurs because multiple unpaired but unused cells of some size may exist and yet be unusable if a larger block of storage is needed. This fragmentation is not a conceptual problem for APL, because active cells can be swapped at any time so that the unused cells do become buddies. It can, however, have some practical effect, because the swapping process can be time consuming for the CPU, and can increase the real storage requirements of the system.

Managing large-scale accountable storage

It is not very practical to extend buddy cell sizes beyond 4K bytes on an IBM System/370, because in most cases operating system interfaces do not provide for storage alignment on any boundary greater than 4K. But this is not a serious problem for two reasons:

1. The number and frequency of large allocations is far lower than for small allocations.
2. Once a large area has been allocated, a great deal of effort normally goes into filling it with data.

Both of these reasons ensure that path lengths for large area allocations are not critical. Any of a number of more conventional storage schemes could be used successfully to provide accountability and reuse of large areas. Indeed, it would be feasible to depend on operating system storage management for these areas. APL systems do not do that at the present time because of a concern about storage fragmentation. If storage should become badly fragmented there would be no practical way to recover when using operating system control. So long as APL controls the storage, garbage collection can be used if necessary.

Along with the work to support buddy system cells, there is also a new scheme for managing larger blocks of storage. Historically such schemes have usually been based on maintaining linked lists of available areas. (Each currently unused area contains a pointer to the next unused area in its group.) Since we were dealing specifically with large amounts of pageable storage we were concerned about the potential paging overhead of traversing such chains to locate a usable area.

The solution chosen was to maintain a bit map of storage blocks. This became feasible because the smallest unit of storage to be managed was a 4K page (2^{12} bytes). All subdivisions of that were managed by the buddy system. Because of operating system limitations, the largest total area that APL could be presented with was somewhat less than 1 gigabyte (2^{30} bytes). This meant that all possible pages could be represented by $2^{30-12} = 2^{18}$ bits. This

is 2^{15} bytes, given an 8-bit byte. Thus a bit map for the largest supported amount of storage could be stored in 32 K, or eight 4K pages, a very reasonable amount of space when dealing with a gigabyte of storage. For workspaces up to 128 megabytes the bit map requires only a single page.

One of the problems that linked list management systems must address is coalescing adjacent free areas. This problem disappears with a bit map, since the bits are stored in virtual storage order. Linked list systems can also simplify the problem by using address order for linking, but this usually makes allocation and de-allocation searches too expensive. With a bit map there is no problem at de-allocation time. The storage address is trivially converted into an index into the table. But locating an available area of appropriate size during allocation is another matter.

Bit maps are used for pages and buddy system cells are used for smaller cells.

This was solved by using a set of 256-byte lookup tables to convert one 8-bit pattern to another. A table is chosen based on the number of pages required for an allocation. Each byte in the bit map is treated as an index into the table. The content of the table entry indicates whether the request can be satisfied from that section of the bit map.

If, for example, a request was made for six pages of storage, the request could be satisfied by either

- Six or more contiguous free bits within a byte
- Three or more contiguous free bits at the edge of a byte with the remaining one to three bits available at the adjacent edge of the adjoining byte

The bit configurations satisfying the first criterion are:

```
00000011 00000001 10000001 11000000
00000010 00000000 10000000 01000000
```

Treating these bit configurations as binary numbers, zero-origin entries 3, 1, 129, and 192, as well as 2, 0, 128, and 64 in the lookup table would need to contain values that indicate the configurations are satisfactory.

The System/370 includes a translate and test (TRT) instruction that can automate the search, so long as unsatisfactory configurations have a binary zero entry in the lookup table. Because of this a convention of putting the one-origin offset of the first satisfying bit into the table was chosen.

Seven tables of this form were generated, to allow searches for up to seven contiguous available bits in a byte. Note that if the search succeeds, both the byte and bit numbers of the desired position in the bit map are known.

When fewer than eight pages are required, a fast search is made for all bits within one byte using one of the above seven tables. If this search fails, a byte-by-byte check of the bit map is used to look for an area crossing two bytes. This check also utilizes a lookup table that is indexed by the bit map bytes, but in this case the indexing is done manually, and the codes within the table indicate the number of bits available on each edge of the argument byte (i.e., the code is treated as a pair of 4-bit numbers). By adding appropriate edge-counts from adjacent bytes, the system can determine whether enough space is available at that boundary.

If eight or more pages are needed, a search is made for a byte in which all eight bits are free. Once such a byte is found, the search is expanded around that byte as needed to obtain more than eight pages. If no appropriate area can be found containing an all-free byte, and if 14 or fewer pages are needed, the same edge search is run that is used for less than eight pages.

The expanded search for more than eight bits is somewhat tedious, but it should be noted that the storage areas involved are always longer than 32K bytes, so the processing cost after allocation is usually much larger than the time spent to locate an available area.

In all of the searches a choice had to be made between a "first fit" (or perhaps "next fit") and a "best fit" rule. Bays's analysis[11] shows that next fit is a poor choice, but does not provide a clear preference between first and best fit. We prototyped an

exact fit scan followed by a first fit scan, but found that for our bit map search routines using first fit alone provided slightly better overall performance. We did not analyze the reasons, but assume it was a combination of the extra CPU cost for a double scan and because first fit develops a set of "favorite pages," or those which are less likely to be paged out.

As with the buddy system, there is a storage penalty for the page-oriented allocations. For blocks up to 8192 bytes (8K) the same usage constraints exist as for buddy cells, and the effective utilization is slightly less than 75 percent. This number rises, though, for larger blocks. The only allocations made to a ten page block, for example, are those requiring more than 90 percent of its space. The usage of large arrays varies greatly among applications, so it is difficult to generalize. It is probably safe to say, though, that for most applications the effective utilization of large-scale storage will be between 75 and 95 percent.

Getting the best of both

The previous discussions of buddy cells and large-scale storage each ended with warnings about limits on effective utilization of virtual storage. To some extent this is a deceptive concern. All storage management systems produce small fragments of storage intermixed with live data, and for most systems the fragments are either completely unusable, usable only at great expense, or usable only for a small subset of the allocation requests. But there are three ways in which this is a very real concern:

- The fragments at issue are all less than one page long, and are all on pages containing live data. Thus in a paging system they always act to increase the number of real pages required to run the application effectively.
- Unlike traditional APL storage management, there is no way to "squeeze" the fragments out of the live data and make them available again.
- Because of the previous point, the unusable fragments would still exist in APL workspaces that are saved. This implies an increase in required permanent storage space as well as additional data transfer while reading and writing the workspaces.

To address these concerns a hybrid scheme was implemented. The workspace is divided into two sections, with a floating boundary between them. Storage to the left (low address end) of the boundary contains densely packed objects managed using traditional garbage techniques. Storage to the right of the boundary is managed using bit maps for pages and the buddy system for smaller cells. So long as enough reusable storage is available at the right end of the workspace, garbage is allowed to collect at the left end. In many cases this will suffice for so long as the workspace is active. When a request arrives that cannot be satisfied, some form of garbage collection is done. One of three alternatives is chosen:

1. If there are enough free pages at the right end to satisfy the request (but they are scattered), and there is more storage available in free pages than in garbage at the left end, then allocated pages at the right end are rearranged so that all free pages are in one group.
2. If there is enough garbage at the left end to satisfy the request, and there is more storage available in garbage than in free pages at the right end, then all garbage at the left end is collected, the dividing line is moved left to the end of the last page on that side still containing data, and the remainder of the collected storage is made available in the page pool.
3. If neither end has enough space to satisfy the request on its own, all unused quickcells are released and then a full garbage collection of the workspace is done. At the end of this process the dividing line is at the end of the last page containing data, and the remainder of the workspace is in the page pool.

The third form of garbage collection is always performed when a workspace is saved. (The page pool is not kept with the saved workspace. Indeed when the workspace is reloaded later the page pool may be of a different size.)

Note that this concept of two storage zones is a simplified form of "generational garbage collection" as recently advocated by Appel,[12] Wilson and Moher,[13] and others.

Comparative performance measurements

A limited amount of performance measurement has been obtained comparing APL2 with and without the storage management changes described in this paper. The results are very encouraging, but should not be over-interpreted. A storage-intensive test function was generated that allocated and ini-

Figure 7 Comparison of CPU times

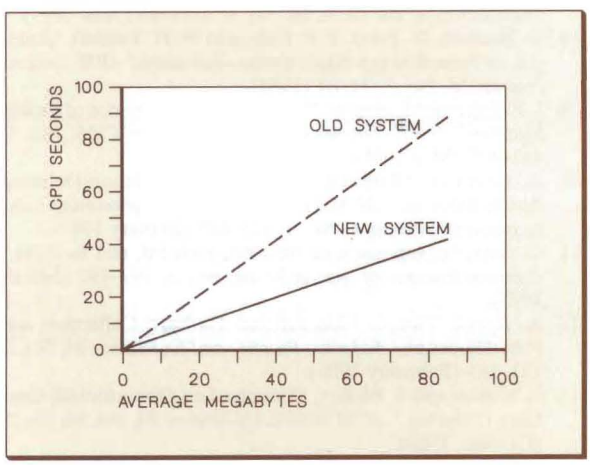
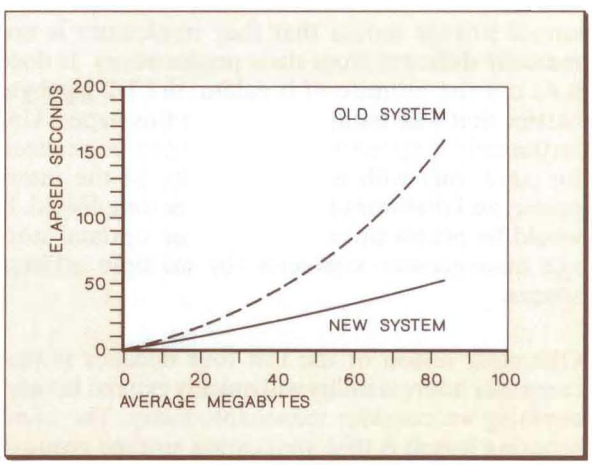


Figure 8 Comparison of elapsed times



tialized storage blocks of random sizes and random lifetimes. The algorithm automatically adjusted to workspace size, and tended to keep an average of 60 to 65 percent of the workspace in use. This is not a typical APL application, but it was created specifically to exaggerate any differences in the storage behavior of the systems.

Figure 7 shows the amount of CPU time used by the test case over a range of workspace sizes. It has been scaled by the average amount of allocated storage rather than by workspace size to remove any bias due to buddy cell internal fragmentation. All tests were run on an IBM 3090* with 128 megabytes of real storage and very little other concurrent activity, so no paging was needed.

Figure 8 shows test case elapsed times from the same runs as Figure 7. We believe that the accelerating slope seen here is caused by IBM's Multiple Virtual Storage Resource Manager function intentionally slowing the application down as larger fractions of the system's total real storage are used.

Testing under loaded conditions produces similar results. As a controlled environment, ten tests were submitted simultaneously and competed for three initiators on an idle system with three CPUs. Separate runs were made with 100-megabyte and 200-megabyte workspaces. With three initiators and three CPUs these resulted respectively in roughly 1.5:1 and 3:1 overcommitments of available real storage. At 100 megabytes the new system used 47 percent as much CPU time and only 38 percent as much elapsed time as the old. At 200 megabytes the elapsed time dropped to 34 percent. In both the 100- and 200-megabyte cases the usable allocated storage dropped by less than 1 percent since internal buddy cell fragmentation is of little consequence in such large workspaces.

Finally, it is important to stress again that the differences shown here are exaggerated from those that would be seen by an APL application. More than 90 percent of the test application time was spent in allocating and initializing storage. It would be more typical for an application to spend between 1 and 10 percent of its CPU time in that code, and it could spend much less than 1 percent of its elapsed time there if it was highly input/output oriented.

Concluding remarks

For 25 years APL systems have depended on garbage collection for storage management, and it has served them well. Pure garbage collection schemes are likely to be used less in the future than in the past, but composite schemes will continue to exist where garbage collection is an important component.

This paper has focused on the current storage management schemes for APL running on IBM mainframe hardware and their operating systems. The issues and solutions would be entirely different, for example, if the storage model used by an IBM Ap-

plication System/400* processor were assumed. This paper has not addressed the unique attributes of Enterprise Systems Architecture systems, but the virtual storage model that they implement is not radically different from their predecessors. It does hold out the promise of breaking the 1-2 gigabyte barrier that was assumed earlier in this paper. Unfortunately it appears the promise can be realized for APL2 only with a major rewrite of the interpreter, and that work has not been accomplished. It would be premature to speculate on optimal storage management strategies for multiple address spaces.

One clear lesson of the last four decades is that computer addressability will quickly expand beyond anything we consider reasonable today. The more sobering lesson is that application storage requirements seem quite capable of expanding as fast as hardware capabilities. This race will not only keep implementers of language products busy for the foreseeable future, it will also keep a noticeable part of their focus on matching these storage requirements and capabilities.

* Trademark or registered trademark of International Business Machines Corporation.

Acknowledgments

Brent Hawks and James Brown did an outstanding job of finding and fixing the many bugs in the original prototype code written by the author to explore this topic. The author would especially like to thank Brent Hawks for the long hours he spent serving as a sounding board for, and generator of, ideas. Finally, thanks are gratefully extended to John Gerth for a thorough review of the text and his very helpful suggestions.

Cited references

1. *VS APL Program Logic*, LY20-8032, IBM Corporation (1976); available through IBM branch offices.
2. D. E. Knuth, "Fundamental Algorithms," *The Art of Computer Programming*, Volume 1, Addison-Wesley Publishing Co., Reading, MA (1968).
3. K. Knowlton, "A Fast Storage Allocator," *Communications of the ACM* **8**, No. 10, 623–625 (October 1965).
4. D. S. Hirschberg, "A Class of Dynamic Memory Allocation Algorithms," *Communications of the ACM* **16**, No. 10, 615–618 (October 1973).
5. B. Cranston and R. Thomas, "Simplified Recombination Scheme for Fibonacci Buddy Systems," *Communications of the ACM* **18**, No. 6, 331–332 (June 1975).
6. K. K. Shen and J. L. Peterson, "Weighted Buddy System for Dynamic Storage Allocation," *Communications of the ACM* **17**, No. 10, 558–562 (October 1974).
7. J. L. Peterson and T. A. Norman, "Buddy Systems," *Communications of the ACM* **20**, No. 6, 421–423 (June 1977).
8. G. Bozman, W. Buco, T. P. Daly, and W. H. Tetzlaff, "Analysis of Free-Storage Algorithms—Revisited," *IBM Systems Journal* **23**, No. 1, 44–64 (1984).
9. I. P. Page and J. Hagins, "Improving Performance of Buddy Systems," *IEEE Transactions on Computers* **C-35**, No. 5, 441–447 (May 1986).
10. A. Kaufman, "Tailored List and Recombination-Delaying Buddy Systems," *ACM Transactions on Programming Languages and Systems* **6**, No. 1, 623–625 (January 1984).
11. C. Bays, "Comparison of Next Fit, First Fit, and Best Fit," *Communications of the ACM* **20**, No. 3, 191–192 (March 1977).
12. A. Appel, "Simple Generational Garbage Collection and Fast Allocation," *Software Practice and Experience* **19**, No. 2, 171–183 (February 1989).
13. P. Wilson and T. Moher, "Design of an Opportunistic Garbage Collector," *ACM SIGPLAN Notices* **24**, No. 10, 23–25 (October 1989).

Accepted for publication July 16, 1991.

Ray Trimble *IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141.* Mr. Trimble is an advisory programmer in APL products development. He has worked in the APL organization since 1976 and has been a technical leader in a number of development projects for both VS APL and APL2. He is the author of the manual *APL2 Programming: Processor Interface Reference* and has been heavily involved in developing other APL manuals. Mr. Trimble joined IBM in 1966 and worked first on a large-scale macro preprocessor supporting multiple languages. Beginning in 1971 he served as joint chief programmer for access methods in the original MVS development project.

Reprint Order No. G321-5446.

Putting a new face on APL2

by J. R. Jensen
K. A. Beaty

APL2/X is an interface between APL2 and the X Window System®, built at the IBM Cambridge Scientific Center. This interface enables the full set of the X Window System Xlib calls and the related data structures to be used directly from programs written in APL2, thereby providing APL2 with a true, full-function windowing environment. The interface also deals with the broader and more general issue of how to call C programs from APL2. The interface and the experience of building it are described in some detail in this paper.

The intent of this paper is to detail the experience of building an interface between APL2 and the X Window System**. APL2, having evolved over two and a half decades, was a good candidate for a "face lift" in that it benefits greatly from having a modern presentation system. In turn, the X Window System gains the flexibility and power of APL2 in developing and driving applications.

This paper is divided into several subsections. To set the stage, some simple examples of how the interface can be used are shown, and an overview of the X Window System is also given. With that as a background, we then discuss the rationale for building such an interface, as well as some of the design choices made. Next, the general APL2-to-C interface that has been implemented is presented. APL2/X uses this interface heavily. The focus is on how to be able to use a large number of already-existing C routines from APL2 with as little additional work required as possible. Finally, examples of how to use this interface to access and call C routines from APL2 are shown, with a focus on the special consideration that the X Window System entails.

It is assumed that the reader has some knowledge of both APL2 and the X Window System. See, for example, *APL2 at a Glance* by Brown et al.[1] for an introduction to APL2, and *Introduction to the X Window System* by Jones[2] for information about the X Window System.

An example. An example might help illustrate the capabilities of the X Window System when used with APL2. To display the image of this example, run the following APL2 expression:

```
XIMAGE MAN COL 'Basic'
```

XIMAGE is an APL2 function that uses the X Window System calls to display an image. *MAN* is an APL2 variable containing an image of a mandrill, *COL* is a color lookup table, and *'Basic'* is a window title. It results in a new window displaying the content shown in Figure 1.

Figure 1 Basic image

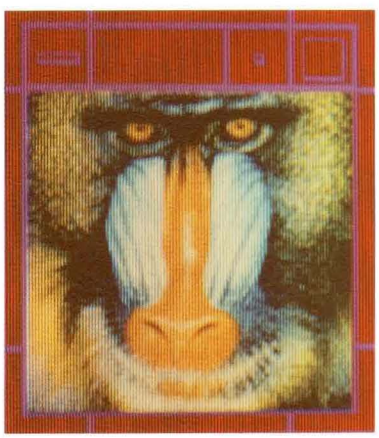
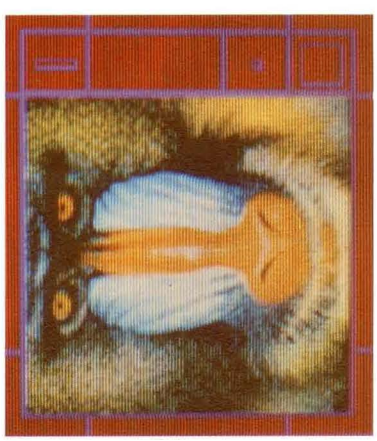


Figure 2 Image turned on side



This image can be manipulated using normal APL2 functions. The manipulation can take place on the image matrix, the color table, or both. For instance, to turn it on its side as in Figure 2 use:

```
XIMAGE (⍉MAN) COL 'Lazy'
```

To triple its size as shown in Figure 3, use the following function:

```
XIMAGE (3/3⌿MAN) COL 'Large'
```

To display as a negative as is done in Figure 4 use:

```
XIMAGE MAN (1000-COL) 'Neg'
```

Finally, to create four mirror images of the mandrill as in Figure 5 use:

```
B2←MAN,⌽MAN
B4←B2,[1] ⊖B2
XIMAGE B4 COL 'Four'
```

Why an APL2 X Window System interface. From the advantages each has to offer, it is evident that APL2 and the X Window System can benefit from an interface connecting them. We now describe some of the more compelling benefits for APL2.

APL2 is provided with a modern-day interface. The present interface of APL2 dates back to the late 1970s and has a distinct character-cell flavor to it. Graphics are limited to fixed, nonmovable images. Several desirable features can be incorporated by utilizing the functions of the X Window System:

- Keystroke sensitivity for programs
- Pointing devices such as a mouse
- Multiple fonts of varying size
- Bitmapped graphics and image
- Dynamic graphics capabilities

Many of these features are as much a product of better hardware (in the form of workstations) as they are of the software, but this does not negate the fact that they need the software to utilize these advanced features.

The interface enables a given APL2 application to display its output on any connected workstation, and enables a workstation to initiate and run APL2 programs on many different hosts at the same time.

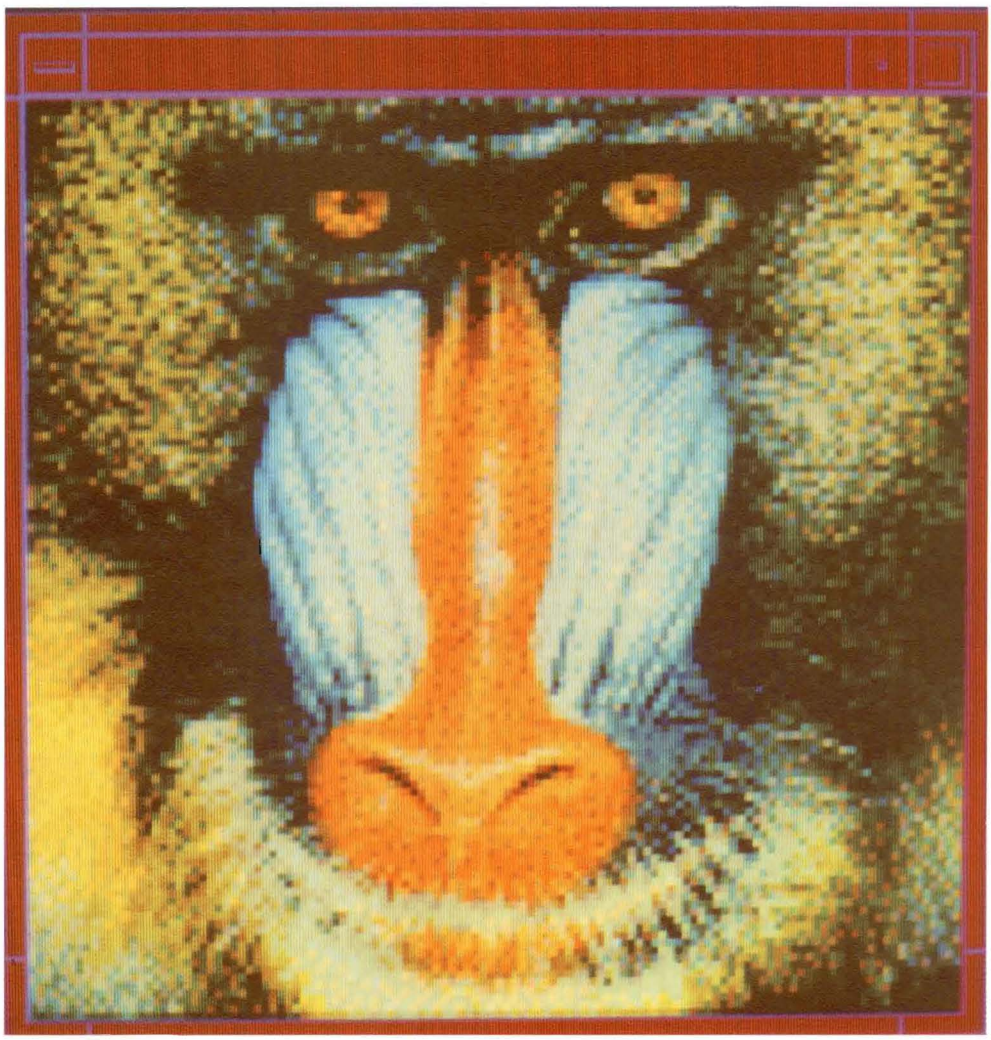
Similarly, the X Window System gains from using APL2. APL2 provides an interactive environment. Each call or series of calls can be tried out, verified, and altered at will until the right combination is reached. This activity can take place without any recompilation whatsoever, speeding up the development process. The X Window System can use the array processing ability of APL2 to easily store and manipulate images using standard APL2 primitives, as shown in the example presented earlier.

For those readers not familiar with the X Window System, the next section presents a brief overview.

An overview of the X Window System

The X Window System is the *de facto* standard for windowing systems in the UNIX** environment. In

Figure 3 Image tripled in size



many respects it is very similar to the Operating System/2* (OS/2*) Presentation Manager* and Microsoft Windows** for the IBM Personal Computer Disk Operating System (PC DOS) in that it provides the application programmer with a multitude of calls to control and manipulate the content of windows on a display. However, it also differs from these products in some key aspects. The foremost difference is that the X Window System was designed from its inception to be network-transparent. This means that an application can display its results on any workstation attached to a local area network, no matter where the application may actually be running. The X Window System employs the client-server model of computing. It enables the application, or client, to make use of the resources of the workstation, or server, to display its output and receive input from the user, as illustrated in Figure 6.

Server. The X Server is a program running on the workstation that manages the interaction with the user. It typically controls one or more screens, a keyboard, and a mouse or similar pointing device. It allows clients to have use of all of these devices and other resources such as windows, pixmaps, fonts, and graphics contexts. The server receives directives from communicating clients via network

Figure 4 Negative of image

protocol requests and acts upon them to draw windows, graphics, text, and images on the display. Whenever the user of the workstation performs an action such as pressing a key, moving the mouse, etc., the server will generate an event message and return it to the client program via the underlying network protocol (traditionally TCP/IP, the Transmission Control Protocol/Internet Protocol).[3]

Client. The application is the client program. It sends requests to the server via the network protocol and receives information back from the server in the form of replies or events. These requests can be generated and handled at the network protocol level, or higher-level calls can be used.

Window manager. The window manager is an application that controls where windows are placed, the size of the windows, the window decorations, and the interaction style. In separating it from the X server, the X Window System has made it possible to have a replaceable window manager implementing different interaction styles. As an example, some window managers enable window moving and resizing by grabbing and dragging the window borders, whereas other window managers will use menus to accomplish the same end result. Among the many window managers that exist, one now in common use is the Open Software Foundation (OSF) Motif** window manager that gives the X Window System a "look and feel" almost

Figure 5 Image quadrupled

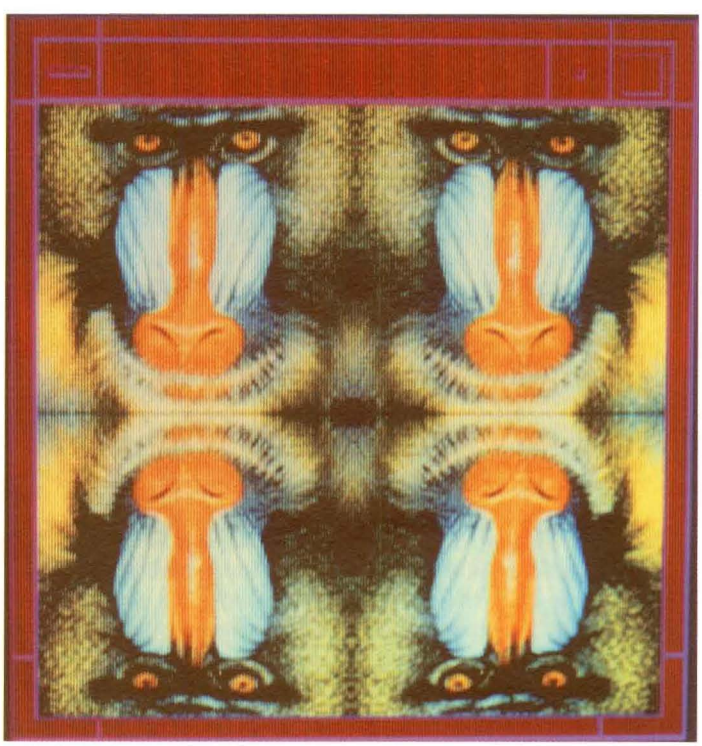
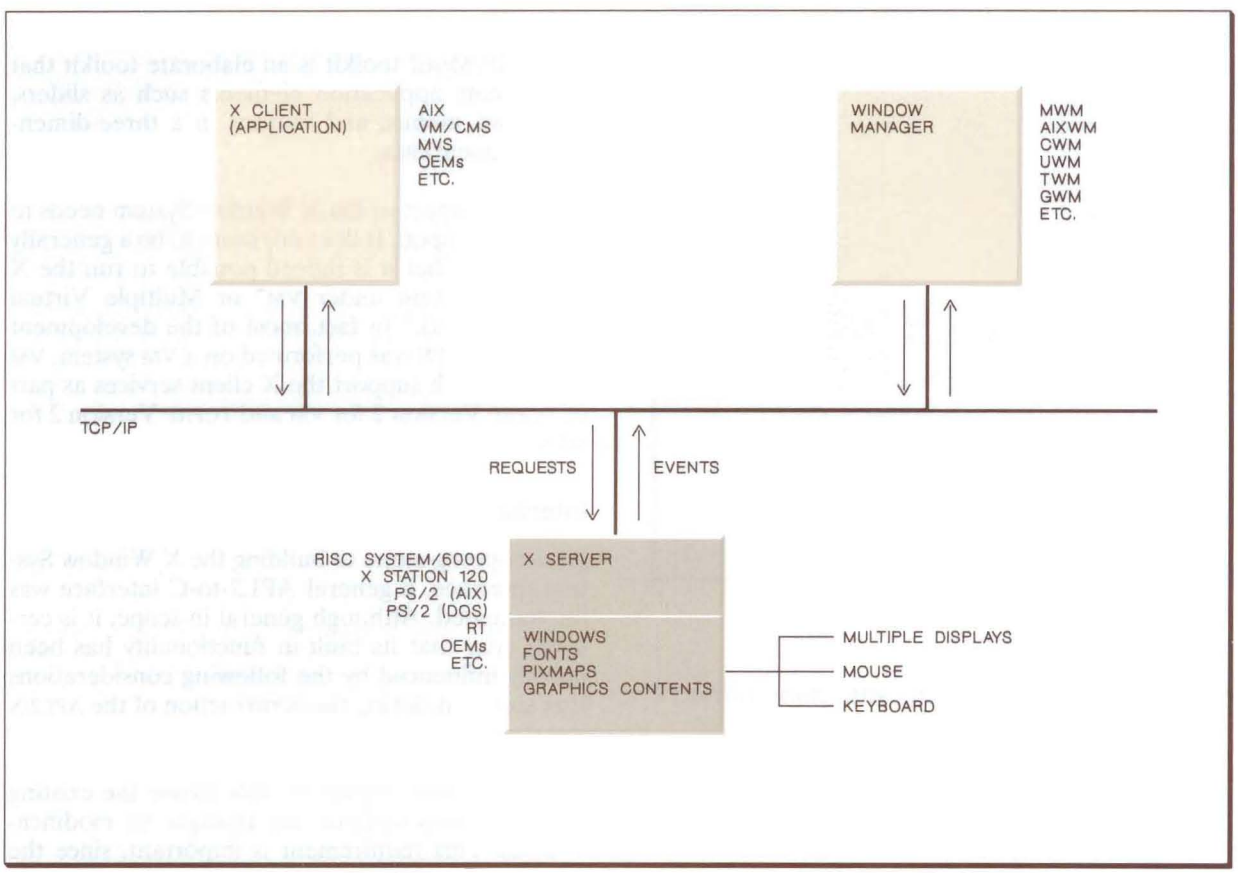


Figure 6 The X Window System client-server relationship



identical to that of the OS/2 Presentation Manager. A list of other existing window managers can be found in Figure 6.

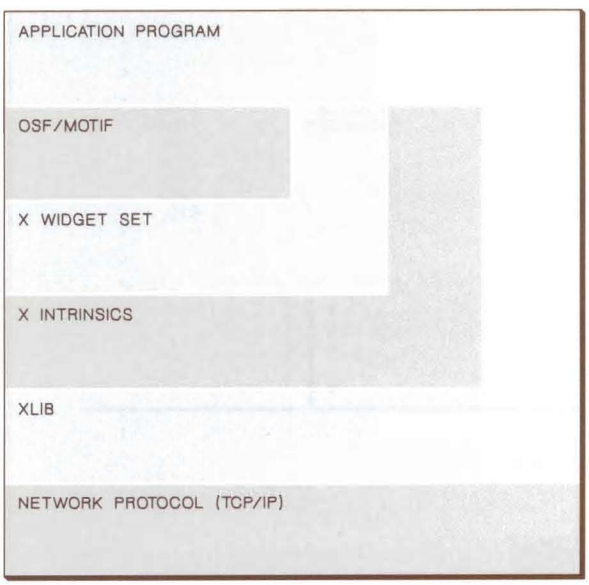
These three components of the X Window System need not run on the same processor. An application can be running on, say, a host with the Virtual Machine/Extended Architecture operating system, or VM, communicating through the X Window System client services with an X Window System server running on an IBM RISC System/6000*. The net effect of this setup is that the results of the application appear on the display of the workstation as though the application had been run locally.

It is also possible for a single client application to display on many servers at once. Likewise, a server can service many clients at the same time, displaying the output of each application program at once. Communications among the client, server, and window manager can be handled by any method that provides a reliable bidirectional byte stream. When the client and server both run on the same processor, some sort of interprocess communication is used for communication between the two. When the client and server are running on separate processors, the TCP/IP communications protocol usually provides this service, although any other reliable communications scheme could potentially be used in its place.

The X Window System output capabilities can be summarized as follows:

- Controlling multiple windows on one or more display screens
- Drawing graphics primitives such as lines, arcs, rectangles, and polygons with or without fills

Figure 7 The X Window System hierarchy



- Writing high-quality text with many different fonts
- Supporting images

Inputs are received in the form of events. Events can be generated by the user pressing a key on the workstation keyboard, by the user manipulating the mouse (moving it or using the mouse buttons), or by other events, such as when a window is cleared and needs to be redrawn.

The X Window System is a layered architecture, depicted in Figure 7. An application can draw upon the calls of all of these layers. The X Window System *network protocol* is at the base of the hierarchy, ultimately defining the traffic flowing between the client and server components.

The *Xlib* level is the next level. Most application programmers will never interface with the X Window System at a level lower than this one. It consists of about 400 separate calls written in C and more than 100 data structures.

The *X Intrinsics* and the *X Widget Set* taken together form the *X Toolkit*. A widget set is a collection of common graphics elements that applications may use, such as menus, scrollbars, pop-up windows, and the like. The X Widget Set makes use of the X Intrinsics, which provides it with an object-oriented interface.

The OSF/Motif toolkit is an elaborate toolkit that implements application elements such as sliders, pull-down menus, and buttons in a three-dimensional appearance.

One final aspect of the X Window System needs to be touched upon. It does not seem to be a generally known fact that it is indeed possible to run the X Window System under VM[4] or Multiple Virtual Storage (MVS).[5] In fact, most of the development work of APL2/X was performed on a VM system. VM and MVS both support the X client services as part of TCP/IP Version 2 for VM and TCP/IP Version 2 for MVS.

Interface design criteria

As a stepping-stone to building the X Window System interface, a general APL2-to-C interface was implemented. Although general in scope, it is certainly true that its built-in functionality has been heavily influenced by the following considerations that surfaced during the construction of the APL2/X interface.

- The interface should be able to use the existing C functions without any changes or modifications. This requirement is important, since the source code for the C functions may not be available.
- The interface must be able to support a large number of calls efficiently. The X Window System defines about 400 separate calls, depending on the release considered.
- To be useful, APL2/X must be able to support data structures but also allow APL2 to manipulate the data using APL2 functions. Data structures play an important role in many X Window System calls.
- The external C routines must be used in a manner much akin to normal APL2 functions (i.e., maintain the "feel" of an APL2 function) when called from within APL2. Specifically, attainment of this likeness requires that function arguments be passed explicitly and by value. The interface must take care of the needed argument type coercion. Also, the interface should specifically re-

frain from using designated variables or storage areas that can be updated as a side effect of the call; rather, all output should be returned as explicit results of the call.

- The defined X Window System call syntax should be adhered to as closely as possible so as to enable the use of the normal X Window System documentation. Only two noteworthy deviations apply throughout the interface:

  1. Output-only arguments are never specified on input; they will be generated automatically and returned as part of the explicit result of the function call.
  2. Arguments are given by value, even in cases such as a character string, where C expects a pointer in the parameter list. The interface again handles the details of making this happen.

  At times, maintaining this fidelity to the X Window System call syntax seems slightly out of place in an APL2 setting. One of the places where this is apparent is on those calls where the X Window System expects a varying number of arguments passed in an array or character string. These calls invariably require the specification not only of the array itself, but also of the number of elements in the array. This latter piece of information is, of course, directly available with the APL2 array, so it seems slightly silly and annoying to have to specify it in the call. However, in the name of consistency we have chosen to stay with the X Window System call syntax throughout, even in cases such as this one.

- APL2 will handle storage management automatically, whereas C most often leaves the task for the caller to do. APL2/X takes over this chore when calling the C functions, so the APL2 program is freed from addressing this task explicitly.

- Enable the same interface from APL2 to the X Window System in multiple host environments to allow APL2 applications that use APL2/X to be run under the Virtual Machine/Conversational Monitor System (VM/CMS), Multiple Virtual Storage/Time-Sharing Option (MVS/TSO), or under Advanced Interactive Executive* (AIX*) on the RISC System/6000.

All of these items are discussed later in more detail.

Using the X Window System from APL2

APL2 can use the X Window System in two different ways. It can either use it indirectly, if the output device APL2 is communicating with is being remapped to a workstation running the X Window System, or directly by issuing calls to the X Window System from APL2. The indirect approach allows existing applications written for a 3270-type display screen to run on an X Window System workstation, but the interaction style is then, of course, limited to that of a 3270 device. However, to be able to utilize the features and facilities of the X Window System, it is necessary for the applications to be given direct, explicit access to the X Window System.

APL2 using the X Window System in compatibility mode. The simplest way today to use the X Window System from APL2 is in compatibility mode. Essentially it is another way of getting someone else to worry about supporting the X Window System. Two existing IBM products that do just that are described below.

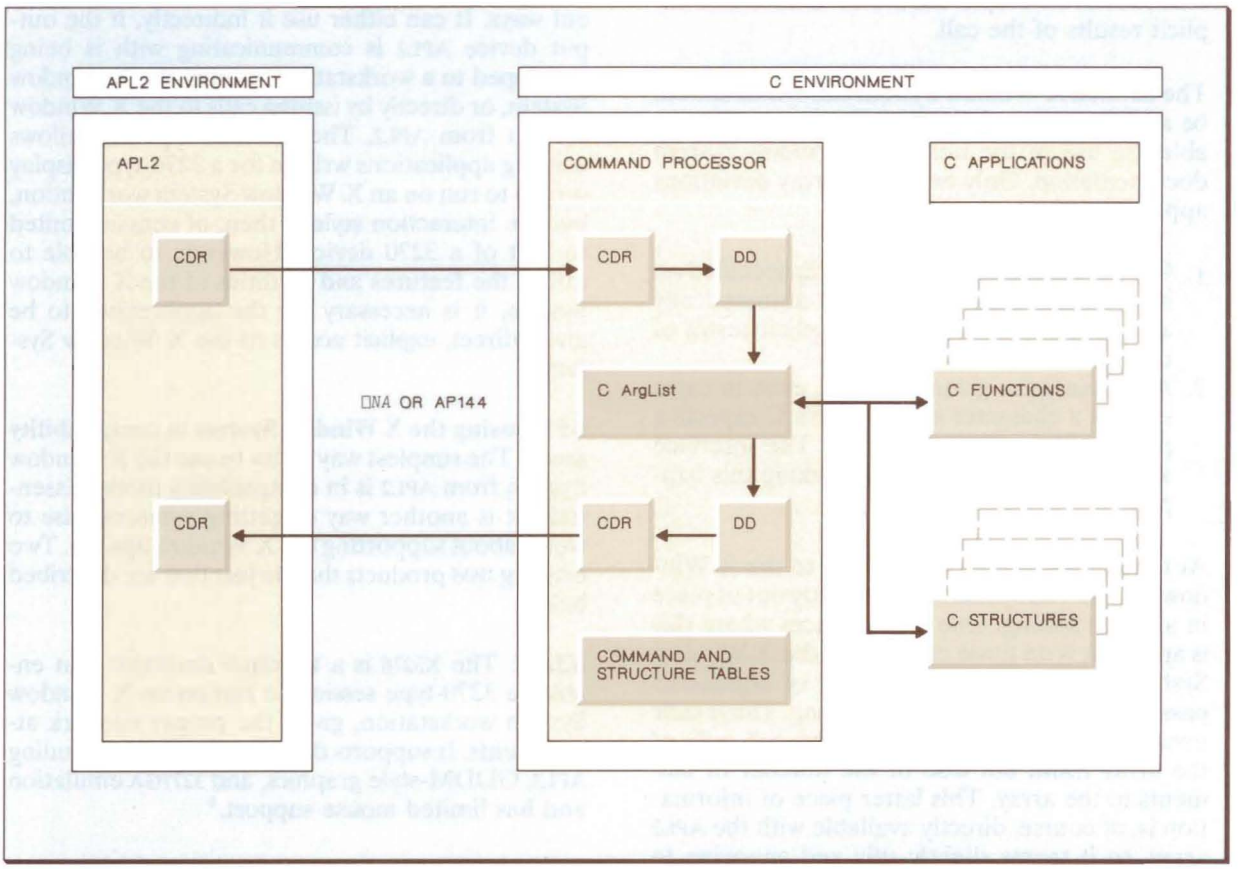
X3270. The X3270 is a terminal emulator that enables a 3270-type session to run on an X Window System workstation, given the proper network attachments. It supports different size fonts including APL2, GDDM-style graphics, and 3277GA emulation and has limited mouse support.[6]

GDDM/XD. GDDM/XD is an interface that permits the display of output from GDDM on workstations supporting the X Window System. It is available as part of TCP/IP Version 2 for VM and TCP/IP Version 2 for MVS. It displays both character and graphics output in a separate window on the X Window System workstation.[7]

Exploiting the X Window System from APL2. APL2/X takes a different approach to the X Window System. In order to fully exploit the X Window System from the APL2 environment, it is essential that the application be given direct access to all of the X Window System calls.

The connection between C and APL2 is illustrated in Figure 8. APL2/X receives data from APL2 in its common data representation (CDR) format. The CDR format is a documented data format for APL2 external data. It includes not only the data, but also descriptive information about data type, rank, and dimensions. The format varies, depending on the

Figure 8 Calling C programs from APL2



host operating environment. The data are sent from APL2 to APL2/X using the associated processor 11 (VM) or a new auxiliary processor AP144 (AIX).

Once in APL2/X, the incoming CDR is transformed into a DD, or data descriptor, which is the data representation used internally by APL2/X in all of the host environments within which it operates. This transformation essentially involves breaking up the CDR into self-contained arrays connected via pointers. This data representation can be used directly for new functions specifically written to use this data structure.

However, it is more common to use already-existing C functions. To do so, the data must be in the form that the functions can use. The second transformation is then involved to build the *C ArgList*. The argument list is for the C function that is to be called. The ArgList format is also employed when accessing and using C data structures.

The conversion process is controlled by a command definition that describes the arguments required by a given command. These definitions are stored in *command tables*. The first argument in any call identifies the command to be executed. The tables are searched to locate the matching command definition.

APL2/X data descriptor

APL2/370,[8] APL2/6000, and APL2/PC[9] all pass data to C in the form of a monolithic block of data. This block includes not only the data, but also information describing the data type, rank, and dimension. APL2/X breaks up the block of data into its component pieces, storing the descriptor and data infor-

mation in separately allocated areas for each nesting level of the data. This division ultimately cuts down on the amount of data copying needed, and has also enabled APL2/X to extend the descriptors with additional information. The descriptors currently hold the following pieces of information:

`rc`	Element return code
`flags`	Assorted control flags
`refs`	C indirection count
`data`	Pointer to data values (or the data value in the case of a scalar)
`alloc`	Number of elements allocated
`xrho`	Number of elements stored
`rtl`	Data type
`rank`	Rank
`dims`	Array dimensions (zero, one, or more)

APL2/X adds the `rc`, `alloc`, `flags`, and `refs` items for its own use. The remaining items are extracted from the data passed from APL2.

The ability to use separately allocated items has proved to be very useful when constructing elaborate return values. As an example, see the result of the `GetConst` command given later in the section on support for C constants. The result consists of a three-column table. The first column consists of a character string, the second column another character string, and the third column a value that can be either numeric or yet another character string. This table is built in a bottom-up fashion, with each element being appended in turn. With use of the separately allocated items, it just becomes a question of keeping track of a set of pointers, whereas a monolithic approach would require a preliminary pass to determine the size of the final table, before the actual building of it could get under way.

Defining and calling C functions from APL2

The supported X Window System calls are defined in a command table, along with their parameter and result type codes. The type codes are used to validate argument inputs and to gather resultant output for returning to APL2. The X Window System command table, as well as the related X Window System structure definitions table, are compiled into the command interface written in C. Therefore, the interface that resides between APL2 and the actual C functions (commands) being called is the one responsible for validating input and checking for and returning expected function results.

By having the command and structure definitions reside in the C command interface, we can funnel all Xlib calls through a common APL2 function rather than having an APL2 function for each X Window System function call. This significantly reduces the number of X-Window-System-related functions that need to be present in the application workspace.

The APL2/X interface ends up being identical in VM and AIX. Two simple APL2 functions *C* and *X* hide the fact that communication between APL2 and APL2/X is handled by processor 11 in VM and by shared variables in AIX, giving APL2/X a single common interface to APL2 in all host environments. These functions with calls and parameter are given in the following box.

```
(rc [results]) ← C command [parm] ...

[results] ← X command [parm] ...
```

The terms in the box are defined below.

command — The name of the X Window System call to be invoked, specified as an APL2 character vector.

[parm] — All but a few of the X Window System calls require additional input parameters to be specified. These parameters are given after the name of the call itself, in the same order as listed in the X Window System documentation.

[results] — The output from the call (if any) is returned in the form of an explicit result. This result includes the X Window System explicit result (if any), as well as any implicit results passed back via output parameters given on the call.

rc — The command return code. Note that this is only returned when using the *C* function. *C* and *X* only differ from one another in the way they deal with error conditions. *C* passes back an error code as part of the function return. It is then up to the calling program to check this code and take appropriate action on a nonzero return code. *X* supplies a default error-handler to check the return codes as they are returned from each call to APL2/X. *X* will suspend operation in the function by issuing a "`□ES 0 1`"

event, if an error is encountered. The programmer then has a chance to correct the problem.

Using these two functions as the base interface allows easy portability of applications from host environment to host environment, without having to change the calls of the application to the X Window System. We have been able to run identical sample APL2/X applications under VM/CMS and AIX on the RISC System/6000 without changing a single line of APL2 code.

The X Window System calls `XOpenDisplay` and `XDrawLines` can serve to illustrate the close correspondence between X Window System calls issued from C and from APL2.[10,11] In C, the calls might look like the following:

```
int points[4][2]=
    {{10,10},{100,10},{10,100},{10,10}};

dp = XOpenDisplay("");

XDrawLines(dp,win,gc,points,4,0)
```

The same calls can be issued from APL2 (via APL2/X) as follows:

```
points ← 10 10  100 10  10 100  10 10
dp ← X 'XOpenDisplay' ''
X 'XDrawlines' dp win gc points 4 0
```

The call to `XDrawLines` obviously assumes that the parameters `dp`, `win`, and `gc` have been set up by preceding calls to other X Window System functions.

Command definitions. The interface can support an unlimited number of C routines. Each routine is defined by a command definition that describes the needed aspects of the call as follows:

- Command name
- Input type codes
- Output type codes
- Address of C function to be called
- Call method
- Two optional parameters that can be used by the command

Some examples of command definitions (all of which implement X Window System calls) are:

```
ACFN2(XDrawlines     ,"IIII2[]II","" )
ACFN2(XOpenDisplay   ,"S"        ,"I" )
ACFN2(XParseGeometry,"S"         ,"IIIII")
```

As can be seen, not all of the command definition fields need be given explicitly. In the above example, only the first three fields are given explicitly. Instead, they are often set implicitly through the choice of the defining C macro. In the case of `ACFN2` above, the call method is a laid-out argument list, and the two optional parameters are not used. Furthermore, the first argument given to the macro defines both the command name and the C function to be called.

The calls can be grouped into different categories. Each category has a defining C macro associated with it to cut down on the number of items that need to be specified explicitly.

In APL2/X, experience has shown that the commands fall in one of three categories:

1. Most commands can be implemented using the standard facilities available in the base interface. In APL2/X we have implemented about 300 commands this way, or about 75 percent of the total.

2. Some commands require some common pre- or post-processing but are otherwise fairly standard. An example is:

   ```
   AXFAS( XGetGCValues, "IIX", "IG",
          &axGCValues)
   ```

   The call runs the X Window System function `XGetGCValues`. This function returns a pointer to a structure of type `XGCValues`. The cover function takes this pointer and resolves it into its constituent values by using the structure class (`axGCValues`) as a guide to what elements the structure contains. Thus, a call to `XGetGCValues` will return the actual values to APL2, not just a pointer. A number of X Window System calls utilize this function.

   We have used this facility extensively during the development of the interface. A lot of the functionality that is now part of the base interface was prototyped in this fashion and was elevated into the base only when the generality was established.

   In the implementation of the X Window System Xlib calls, 65 calls fell in this category, or 17 percent of the total.

3. The third type of calls consists of the ones that for some reason or another require some specialized pre- or post-processing, e.g.:

```
ACFA0("XNextEvent", "I", "G", axNextEvent)
```

There are a number of reasons why functions end up in this category. Examples are calls returning the X Window System event structures. We use a cover function to convert the event structure pointer to its constituent values, so that the values can be returned by the call, instead of a pointer to the values.

Of the total, 30 X Window System calls required handling as special cases, or about 8 percent.

Command tables. The command definitions are grouped together in tables. For example, all of the X Window System calls are defined in a single table. Typically, a table contains only related commands, although this is not a requirement. These tables form an integral part of the APL2/X interface.

When the *C* or *X* functions are called and the interface gets control from APL2, the interface assumes that the first argument given is the command name. The interface uses this name to search through its command tables looking for a matching command definition. If one is found, it controls any further parameter verification that needs to take place before the actual C function can be invoked.

The default is for the command name matching to be case-sensitive, but it is a matter of a compile-time option to change this default to be case-insensitive.

The command tables not only allowed us to group logically related commands together but also proved to be beneficial during the development stages, where a given set of commands could be worked on by an individual without any fear of overlaying someone else's work. Not all of the tables need be active all of the time; they can be activated and deactivated under user control, and their ordering (governing the command search order) can also be changed.

Currently the APL2/X interface defines the following command tables:

- Xlib calls
- Structure support (structure commands)
- Interface control (system commands)

Tables implementing other collections of C functions and structures can easily be added to this list.

The command tables can also be set up and used in a nested fashion, i.e., subcommands may be specified in a secondary command table. In that case, the command as given by the user is effectively made up of two (or more) separate character strings, one for each command table used. We use this facility to implement some of the APL2/X system commands, but we have not found it that useful overall.

Type codes

The specified command determines what additional parameters need to be given, as well as what information will be passed back as a result of the call. These requirements are described by a series of "type codes" attached to the command, with each parameter described by a single type code.

A large number of type codes have been defined. They are specified using one- or two-character alphanumeric strings. Whenever possible we used the same choice of character codes as those given in *APL2 Programming: System Services Reference* for the APL2/370 processor 11 argument patterns.[12] The code is given on the left side, and its definition follows to its right.

`B1` One-bit Boolean
`B8` Eight-bit unsigned integer
`C1` Character (one-byte)
`E8` Double (eight-byte) floating-point real
`I2` Short (two-byte) integer
`I4` Long (four-byte) integer

Type coercion may be applied by the interface to convert the APL2 data to the type expected by the C function and to convert results from the C function to a type that can be handled by APL2.

To enhance portability we also added definitions that left the actual length of a parameter up to the host environment, e.g.:

`I` Integer—This code can be used whenever a C "int" is called for.

Other additions were called for by specific needs of C and X:

`S` A NULL-terminated character string
`P` A C pointer—This code is treated as a large number that the calling APL2 application program probably should never change.

`X2` Two-byte hexadecimal value
`X4` Four-byte hexadecimal value
`X` Two- or four-byte hexadecimal value, depending on the underlying environment

The hexadecimal values can be specified on input as a bit-vector, as an integer, or as a string of hex characters.

Finally, a couple of special type codes:

`G` Accept any parameter given—This code will often be used where further verification of the input will be performed later. An example can be found in the structure commands. Validation of the content of the structure instance is postponed until the proper structure class definition has been determined.
`_` A place-holder—The value is ignored.

Argument indirection is specified in a C-like manner by prefixing the type code, e.g.:

`*C1` A string of characters
`**I` A double indirect reference to an integer

Arrays are also specified in a C-like manner, e.g.:

`I2[3]` A vector of three (short) integers
`C1[]` A vector of characters—The length is left unspecified, so any length will be accepted.
`I[2;2]` A two-by-two array of integers
`I[2][2]` Another way to specify the above two-by-two array

Some considerations pertaining to arrays:

- Any array passed to a C function is passed as a pointer to the values, not the values themselves, as required by C.
- One or more array dimensions can be left unspecified. The length will then be set according to the incoming data.
- The type code specification is more compact than the one used by the APL2/370 argument patterns and also more like native C and APL2, we believe.

Structures are catered to as well, e.g.:

`{II}` Any combination of type codes can be specified inside the braces, including nested structures.

The type codes are also affected by prefix and suffix modifiers. The prefix modifiers are:

`<` Input only
`>` Output only
`|` Input/output
`?` Optional parameter

The suffix modifiers are:

`...` Repeat last type code as many times as needed to account for the given input values.
`*` Ignore any input parameter beyond those already verified.

Both of these suffix modifiers may only be specified at the very end of a list of type codes or following the last item before a "}" ending a substructure definition.

Parameter passing

All parameters are passed explicitly to and from APL2. APL2/X does not cater to side effects such as update-in-place (i.e., changing the value of an APL2 variable other than by explicit reference), nor does it use call-by-name, where the name of a variable to be used or changed is passed as a parameter and the interface reaches back into the workspace to access the specified variable. Although both are technically feasible to do, there has been neither the need nor the desire to use them. In fact, a conscious effort has been made to stay away from them, as it was viewed as detrimental to the clarity of the resulting code.

On calling a C routine from APL2 only those parameters listed as "input" or "input/output" must be specified (i.e., the parameters listed in the input type code field). The interface will generate whatever output parameter place-holders are needed in the actual call to the C function. Upon completion of the C routine, all parameters listed as "input/output" or "output" will be returned to APL2, in addition to the explicit C function result (if required). We thus take advantage of the ability of APL2 to return multiple values in the explicit result of a function invocation. This is an outgrowth of the desire to avoid relying on (hidden) side effects.

Many X Window System calls return more than one result via their parameters. The parameters used in this fashion are always identified by including the suffix "_return" with the parameter name. These

parameters appear at the end of the parameter list. We have taken advantage of this fact in the way that the input and output type codes are specified in the command definitions.

`XParseGeometry` is an example of a call returning multiple parameters. The C function prototype and an example of its use via APL2/X are given below:

```
int XParseGeometry(string, x_return, y_return,
                   width_return, height_return)
  char *string;
  int *x_return, *y_return;
  int *width_return, *height_return;
```

```
⍝ Multiple output.
     (mask x y width height) ←
        X 'XParseGeometry'  '25×80+10−10'
```

Parameters passed by value

Parameters are passed to and from APL2 by value. This is true no matter what level of indirection is needed by the C routine to be called. The burden of setting up this activity and administering the space is handled by the interface. Thus, using the `XParseGeometry` example given above, "string" is given as "'25×80+10−10'" in the call from APL2, and the interface will convert this string to the proper "char *" format before calling the real C routine.

Passing the parameters in this fashion maintains the feel of an APL2 function. The housekeeping chores of managing the temporary storage fall upon the interface, not the user.

These statements do not imply that C data pointers are never returned to APL2, or are used by it. Quite the contrary, pointers are typically specified using the "P" or "I" type codes and are passed back to APL2 as large numbers. The application running in APL2 may use this large number on subsequent calls to external functions via APL2/X but will rarely, if ever, have a need to modify the value of the pointer.

Dealing with structures warrants some special comments. We prefer to pass them by value, and given a choice we have set up the calls to do so. However, there are enough exceptions to this procedure to prevent it from being a general rule. The exceptions come about for the following reasons:

1. Performance—It is inherently more expensive in processing time to create the structure on the fly from its values. If a structure instance is being used repeatedly without its content being redefined, it is more efficient to create the structure once and then refer to it using the pointer to the created structure instance.
2. Permanence—The structure may be modified by future calls. It is therefore important that it remains in a fixed location in storage.
3. Hidden side effects—Most X Window System structures do not exhibit this problem, but we did encounter it using the "Xrm" class of calls. Although unstated, the structure pointer was also being referred to in a hidden lookup table. Another manifestation of such effects is where a data structure has an unspecified or hidden prefix or suffix section.

Wherever possible, APL2/X allows structures to be specified on input either by value or by a pointer to an already-existing structure instance.

Support for C data structures

As would be expected of any sizeable C application, the X Window System defines close to 100 C data structures. Therefore, to fully support the X Window System, the APL2/X interface had to be able to provide access to these data structures as well as the many function calls that are defined by the X Window System. In doing so, APL2/X has implemented these data structures in C and provided import and export access from APL2.

Those familiar with the object-oriented paradigm will recognize the similarities in that approach to the APL2/X handling of data structures. APL2/X maintains a structure (class) definition as part of the C command interface. APL2 calls upon this definition to create new instances of the structure in memory and to assign values to and retrieve values from the fields (class data members) of the instance.

The structure instances are stored in memory controlled by C and thereby directly available to the C application, in this case the X Window System. Upon request from APL2, the instance of the data structure is mapped to an APL2 vector. The vector may be simple (homogeneous) or general (heterogeneous), depending on the underlying C definition. When in APL2, the array can be manipulated in the normal APL2 fashion.

Structure commands. A common set of structure commands has been defined to allow APL2 to easily

create and access the data structure instances maintained by C. Again one can draw comparisons to these structure commands and those implemented for class definitions in many object-oriented languages. The structure commands provide the means to create instances of a given structure type, to perform the chores of getting data in and out of it, and to free up the space once it is no longer needed.

Listed below are the commands that are defined. The commands are shown in three groups: those in the left column operate on a single instance of a structure, the commands in the middle column operate on multiple adjoining structures, and the ones on the right return assorted information from the structure definition.

```
Clear          MClear          GeConst
Get            MFree           GetFields
New            MGet            GetSize
Put            MNew
NewPut         MPut
SFree
```

Structure command usage. The syntax common to all of the structure commands includes the command name followed by the structure type. For those commands that deal with existing structure instances, the pointer to the structure instance (its handle) is expected as the third argument. Following is the general structure command syntax as called from APL2:

```
(rc [result]) ← C command struct [parm] ...
```

Some examples of using these commands are:

```
⍝ Create a new XTextItem instance
      item ← X 'New' 'XTextItem'

⍝ Now fill it with data
      X 'Put' 'XTextItem' item
         ('Simple' 1 2 3)

⍝ Verify that the data made it in
      X 'Get' 'XTextItem' item
 Simple 1 2 3

⍝ Use the structure in a call
      X 'XDrawText' dp w gc x y item 1

⍝ Remember to free it when all done
      X  'SFree' 'XTextItem' item
```

Structure type definitions. The structure type definitions are grouped in tables in the same manner as are the command definitions. In fact, the APL2/X interface provides for each environment grouping to accommodate both a command table and a structure table, as these definitions often go hand in hand. Currently, these three structure definition tables are provided by APL2/X: X events and other X structures, and C primitive structures.

In order to have the structure commands work, the tables must specify the structure type being addressed. The structure type located in one of the predefined tables provides the definition of the elements of a structure instance of that type. Specifically, the type definition contains information about each field of the structure, the names, and data types. The field data types are specified using the same type codes as are used for the function arguments in the command tables.

This structure definition information is also readily available from APL2 via the interface. Having this information available can be of great assistance when using the data structures from within APL2, in that it associates each element in the vector with its related field name in C.

To help illustrate the point, this is how the XTextItem structure from the X Window System Xlib.h header file is defined in C:

```
typedef struct {
  char *chars;  /* Pointer to string        */
  int   nchars; /* Number of characters     */
  int   delta;  /* Delta between strings    */
  Font  font;   /* Font to be used, or None */
} XTextItem;
```

APL2 accesses the structure information in the following manner:

```
⍝ Get all XTextItem fields
      X 'GetFields' 'XTextItem'
 char *chars S
 int nchars  I
 int delta   I
 Font font   I
```

Note that the full C definition of the field is maintained even though the field name and the field type code are the only pieces of information used by APL2/X. The C data type specification (e.g., char *, int, or Font) is kept as part of the field definition since it is often very useful, if not crucial, to the understanding of the role of a given structure member.

Structure field access. To accomplish the equivalent of field access by name as provided by C, two APL2 functions, *axGetFF* and *axGetFF1*, are included as part of APL2/X. These functions use the field information provided by *GetFields* to associate indices to the various field names, thereby providing the index-by-name capability for a related structure instance held in an APL2 vector. The main difference between these two functions is that *axGetFF* provides the indexing for all of the fields in the structure, and *axGetFF1* returns index information for selected fields specified in the call.

By means of an example, we now demonstrate how the `chars` field of an `XTextItem` structure instance, `text`, is accessed from both C and APL2. Note that the *axGetFF* function has previously been called in APL2 to associate the correct index to the field name:

In C:	In APL2:
`text.chars`	`text[chars]`
`text->chars`	`text[chars]`

As a benefit of obtaining the field indexing of the structure from C, the APL2 application can have a measure of independence from changes in the order of fields in the underlying C data structure. That is to say that as long as the fields remain intact and the C structure definition is maintained in accordance with the C application, the APL2 application will not have to change either.

Abandoned approach. Originally we implemented the structure support using "typed" instances so that each instance had a hidden header section that identified the structure type. This implementation meant that the structure class did not have to be specified on each structure command since the information was already available. However, when the structure was allocated by the C application instead of the APL2/X interface, it meant a lot of extra work because the interface would have to allocate another instance with the proper header attached and then copy the structure data of the application into this new area. With the implementation of nested structures this activity became difficult to control, so we ultimately abandoned the "typed" instance approach.

Support for C constants

If the X Window System defines a large number of structures, it defines ten times that many constants in its header files. As any experienced programmer would attest, the use of constants is a major benefit to an application in that it provides symbolic reference so that when a change is called for, only the constant value needs to be changed, regardless of how many references exist. Because these constants disappear during the compilation process, there is no penalty for defining large numbers of them, and the X Window System takes advantage of this and defines a large number of these constants in its header files.

The sheer number of constants employed by the X Window System dictated that APL2/X implement access to these constants in a selective manner rather than expose the whole lot. This approach is logical since any given constant is typically used by only a very limited number of structures or functions. In fact, in the majority of cases, the constants defined in the X Window System header files are related to specific fields of a structure. Therefore, in giving APL2 access to these constants, the constants are logically tied to a related structure definition.

The `GetConst` command provided by APL2/X as part of the structure commands is used to retrieve the constant values associated with a given structure for use in APL2. Following is an example of the output from this command:

```
      X          'GetConst'      'XSizeHints'
USPosition           X                 1
USSize               X                 2
PPosition            X                 4
PSize                X                 8
PMinSize             X                16
PMaxSize             X                32
PResizeInc           X                64
PAspect              X               128
PBaseSize            X               256
PWinGravity          X               512
PAllHints            X               252
```

It is a simple task for an APL2 function to issue this call, create a set of variables, and initialize them to the constant values that are returned. In fact, the *axGetFF* and *axGetFF1* functions previously introduced in the last section not only define structure field indexing, they also create these constant variables for use by the APL2 application.

By doing so, the APL2 functions are able to use the same constants as defined by the X Window System. Such usage insulates the application from changes to these constant values. We experienced

an example of this when upgrading the APL2/X interface support of the X Window System from release 11.3 to release 11.4. Release 11.4 had changed some of the constants associated with the `XSizeHints` structure, among other changes. These changes meant that the table holding the constants in APL2/X had to be recompiled to pick up the changed values, but through the use of the *axGetFF* function it never affected the APL2 applications.

System commands

APL2/X provides a group of system commands in addition to the structure and X Window System commands. These commands are used to control and interrogate the interface itself, as opposed to accessing and using external functions that supply the application with needed services. The names of these commands all start with a closing parenthesis, mimicking the APL2 system commands.

The following system commands are presently defined:

`)Cmds`	List the available commands
`)Env Get`	Get current command environments
`)Env Set`	Change the command environment order
`)RC`	List a return code message
`)Structs`	List the available structures
`)Syntax`	List the syntax of a specific command
`)Version`	Return the APL2/X version identifier

Some examples of their use follow:

```
      X ')Syntax' 'XParseGeometry'
Xlib  XParseGeometry  S  IIIII

      X ')Version'
APL2/X Development Version 0.00

      X ')Env' 'Get'
Xlib  Structs  System
```

Return codes

A major difference between APL2/X and processor 11 of APL2/370 is in the way that errors are reported. Processor 11 treats this condition at an atomic level, using the normal APL2 error messages such as *DOMAIN ERROR* and *VALUE ERROR*. If the error stems from using an element of the wrong type in a vector of arguments, it can be quite difficult to locate the source of the error, especially since the APL2/370 *⎕NA* argument pattern information is not directly available to the application.

APL2/X improves error reporting in several ways. First of all, the arguments in error can easily be determined, since each argument passed to APL2/X will be associated with a return code. Second, the return code is tied to an error message explaining the source of the error, if using the *X* APL2 function. Third, the syntax of the call is available for inspection via a system command.

For instance, using the *X* function:

```
      X 'XOpenDisplay'
Error in input (RC=1)
 Index rc parm
     0  0 XOpenDisplay
     1 16
 16 Expected parameter of type '%s' is missing
Command 'XOpenDisplay' defined by:
   Xlib XOpenDisplay SI
      X 'XOpenDisplay'
      ^
```

Note the use of the default error handler that is part of the *X* function; it will halt execution at the place of error and will point out the parameter or parameters in error.

Using *C* instead (without the trailing comment, of course; it is just placed here for explanation):

```
      C 'XOpenDisplay' 'first' 'second'
 17 0 0  17
⍝ 17: Too many parameters
```

The *C* function does not halt the processing when an error is encountered. Instead, it returns a nonzero return code to the application, and it is up to the application to take whatever corrective action is required. Note the structure of the element return codes: it contains an element for each given or required parameter, whichever count is the larger of the two. This way it is possible to uniquely identify the source of any errors in the parameters.

This principle extends to nested parameters as well, as the following example shows:

```
      C 'Put' 'XTextItem' 509120
        (1 'text' 2 3 4)
 1    0 0 0  26 21 0 0 17
⍝ 17: Excessive number of parameters given
⍝ 21: Dimension 90i must be equal to 90s
⍝ 26: Cannot convert from type 90 to type 90s
```

Issues in calling C routines from APL2

The initial version of APL2/X was completed on a VM system, using processor 11 of APL2 Version 1 Release 3 to call functions external to APL2 itself. The only two programming languages specifically mentioned in the documentation for processor 11 are FORTRAN and System/370 Assembler.[13] Initially we used the FORTRAN linkage-type of processor 11 rather than OBJECT. It was chosen because it would include the length information for each parameter passed. However, in trying to call routines written in C, we encountered the following problems that had to be solved in order for us to implement the X Window System interface:

- Character strings not null-terminated—Character strings are by definition required to be terminated by a null byte in C, but processor 11 does not ensure that the strings passed are null-terminated.
- Returning the result of a C function to APL2—C functions compiled with the C/370 compiler place the result in register 1, but processor 11 expects a result to be passed back to APL2 in register 0.
- Using C pointers—It is not possible to specify a given parameter as being a pointer, such as the C definition `char *` would require. The argument patterns[14] of processor 11 do not cater to this type of definition, and it is therefore possible to handle the distinction of passing a parameter by value, as opposed to passing it by reference.
- Fully specified function argument patterns—The function argument pattern of processor 11 must be completely known by the time a function is called. It is not possible to defer processing and verification of some of the arguments until later, or to ignore others altogether. Thus, it is not possible to call a given function with differing types of arguments.

The above problems are related to calling a single C function. In addition to these problems, trying to implement an X Window System interface introduces another set of problems related to the sheer number of calls to support (395 in the case of the X Window System):

- No list options—There is no call to obtain the function argument pattern of a given external function from within APL2 (short of extracting it from the names file), or to obtain a list of all the accessible external functions.
- Cumbersome to implement and maintain—For a function to be used, it must have an entry in both the names file and the assembler stub module, as well as a $\square NA$ definition in the workspace. Each workspace needing access to the X Window System therefore ends up with a large number of function definitions, in most cases swamping the real functions of the application.

As can be seen, most of these problems revolve around parameter passing. They have been solved in APL2/X by having the interface itself take over the parameter verification chore, using the FUNCTION linkage-type of processor 11, without any parameter verification imposed by the processor. And instead of storing the argument patterns in an external names file, APL2/X now stores these patterns in command tables internal to the interface. Thus, what APL2/X receives is the APL2 data specified by the calling function, and it is up to the interface to perform any needed parameter validation and coercions. This scheme has given APL2/X maximum control of the parameter passing, and thus the following results have been achieved:

- Only a single external function is established in the workspace. The name of the C function to be called is now passed as the first argument in the call.
- Null-termination of character strings is handled automatically by the interface. It avoids having the caller do it in APL2 by either imposing a fixed-length restriction on each string or requiring that the string include the `NULL` terminator.
- The interface supports pointer variables. The support caters to an unlimited number of reference indirections. As an example, an argument with a declaration of "`int **`" is supported. This would be specified as "`**I`".
- Argument verification has been extended to allow for deferred verification. Such verification has proved to be especially important when working with data structures, where the content and structure can vary greatly from structure to structure.
- Additional data types are supported, such as hexadecimals. Also, some data types can be specified in multiple ways. An example of the latter is a bit-field, which can be specified as a vector of bits, an integer (i.e., packed bits), or in hexadecimal format (in the form of a character string).
- Multiple results can be returned as explicit results of the call to a given C function, without the need to build special APL2 functions that preallocate variables to hold the returned information.

- Commands have been added to interrogate the interface itself. This interrogation enables the interface to return the expected syntax of a given call or provide lists of the commands and structures supported.
- Using function linkage has enabled APL2/X to use the processor 11 service routines. These routines provide some useful services, such as data conversion and execution of APL2 expressions from within C.
- A large number of utility functions have been implemented in C that allow us to process and manipulate APL2 data structures in C in an easy and proficient manner.

Taking over the argument verification job turned out to be a blessing in disguise for APL2/X. It made the "port" to the APL2/6000 and APL2/PC environments very easy to accomplish. (In APL2/PC, only the basic APL2-to-C interface has been implemented, not the support for the X Window System.) Both of these environments communicate with APL2/X via a shared variable interface, unlike the APL2/370 implementations. Except for different internal formats of the APL2 data passed from APL2, the processing remains the same as far as APL2/X is concerned, at the internal level and, more importantly, at the user interface level too.

Changes to the X Window System call syntax

One of the design goals for APL2/X was to implement as faithful a representation of the X Window System in APL2/X as possible. However, some differences exist due to the very different nature of C and APL2. The important differences are:

- Function arguments are always specified by value in the same way that they would be for regular APL2 functions. This is true for all types of arguments, scalars as well as arrays. APL2/X performs any needed type coercion and also adds any required indirection pointers based on the type code information before making the actual call to the C routine.

  Note that the explicit use of pointers in APL2/X is not precluded. In fact, they are used as such in many of the X Window System calls, as well as in the routines that implement the structure calls. In these cases, the pointer given is a "magic" constant; as far as the application is concerned it is a value that uniquely identifies some available resource, and no explicit changes to the value should be attempted. A prime example of such a constant is the X Window System "display pointer." This pointer is used on most X Window System calls, but no calculations are ever performed on the pointer itself.

- Only input parameters may be specified on the calls to the X Window System. APL2/X automatically adds any needed output parameters that the call may require. It is a change from the C environment, where the output parameters must be specified explicitly on the call and space possibly allocated to hold the results.

- All results from calling a function are returned as explicit results, including results returned in C via changes to the output arguments. No side effects such as changing of global variables in the workspace are employed. Also, pointer arguments are de-referenced, so what is returned in APL2 are the data values, not the pointers.

The above differences are a consequence of the basic design philosophy underlying the APL2-to-C interface. Another difference, described next, is specific to the X Window System calls dealing with event structures and is more a matter of convenience.

X Window System events are always set or returned by value. The event data are then immediately available for use in the APL2 environment, instead of the structure commands being employed to retrieve the event structure values on the basis of a returned pointer value. The rationale for this decision is that the event data are almost invariable as required by the APL2 application, not just the event pointer, so APL2/X returns the data to speed up the process. In the rare cases where the event pointer is required, it can be acquired through a separate, special call to the interface.

Potential improvements

Although we have come a long way in providing APL2 with access to the X Window System, more work can certainly be envisioned. First among the possibilities would be to add a layer of APL2 functions to help use the X Window System facilities. This could shield some of the complexity of the X Window System, in much the same way it was done in the past in the workspace FSC126 that helped APL2 create and use a full-screen panel by accessing routines of GDDM via APL2 cover functions.

A second option is to extend the range of supported X Window System routines. The Xlib layer is the only layer supported today. There appear to be no technical problems in extending the support to higher layers of the X Window System functionality. It is certain that APL2 would benefit from gaining access to higher-level routines that create and manipulate window system items such as menu bars, sliders, pop-up windows, and other items associated with a modern, windowed user interface. Indeed, this possibility is not restricted solely to the X Window System libraries; other collections of C functions can be accessed equally well from APL2 via this interface.

In an effort to improve the interface for use with C applications in general, some experimental work has already gone into providing the means to dynamically define C commands and structures from APL2. This capability allows APL2 to directly interface with existing C applications without requiring the definitions of the related functions and structures to be built into the APL2/X interface itself.

Last, an even better support for data structures is possible if implemented in APL2 itself, maybe in the form of an option on □*NA* to allow APL2 to access external data variables in much the same way that external functions today are supported. An advantage would be a single copy of data, with the obvious corollary of improved data integrity.

A final example

It would appear as though it is a rite of passage for a windows-based system to have a "HelloWorld" sample program. APL2/X follows this trend. The HelloWorld APL2 function listed in Appendix A illustrates how many of the concepts and ideas presented in this paper fit together. It shows how an APL2 function can implement the two fundamental concepts of a windows-based system: window manipulation and responding to user-generated events. We will let the function listing speak for itself as to the detail; for a more in-depth discussion of the program, see *Introduction to the X Window System*, Chapter 2,[15] *IBM AIX APL2/6000 User's Guide*,[16] or *An Interface Between APL2 and the X Window System*.[17]

Note that the function as listed takes a simplistic view of the world. It has only minimal error-checking, and it is coded as a single, large function. A production-level version of the same function would certainly have to do a more thorough job verifying that error conditions had not occurred. Also, much of the functionality would be implemented through secondary functions common to many windowed applications. However, since the focus of this paper is purely and solely on the capability to access and call C and the X Window System routines from APL2 functions, this example is presented in the form given.

Summary

A major goal achieved in this project was to enable APL2 to use the exciting new facilities that the X Window System embodies and to bring to the X Window System the power of the APL2 interactive environment and array-handling capabilities. This truly brings the potential of a modern-day interface to APL2 while at the same time augmenting the X Window System. A second goal was to provide a common interface to the C language from all of IBM's APL2 systems, ranging from PC DOS through AIX on the RISC System/6000 to VM and MVS, including full support of C data structures. A third goal was to maintain the function-call "feel" of APL2, enabling the external functions to be used as though they were truly written in APL2.

To achieve these goals a number of large issues had to be overcome. Among the more daunting ones were data mapping, handling storage management, and automatic parameter indirection so vital to any C interface. Since APL2 and C are so diverse in the way they deal with storage management, it proved to be a real challenge, especially when dealing with data structures.

The APL2/X interface described is currently available to IBM customers on two APL2 platforms. In APL2/6000 for AIX on the RISC System/6000 (Program No. 5765-012) it is included as the AP144 auxiliary processor,[18] and it is provided as a sample offering with TCP/IP Version 2 for VM (Program No. 5735-FAL) to be used by APL2/VM.[19]

Acknowledgments

We would be remiss if we did not acknowledge the significant amount of help we have received from many people. A special acknowledgment goes to Bob Cohen, who worked with us part time while pursuing his Ph.D., for implementing and verifying a good portion of the X Window System calls. We would also like to thank our manager, Love Sea-

wright, and our center manager, Dick MacKinnon, for making it possible for us to engage in this project; Andy Pierce for showing us his REXX-based interface and providing us the X Window System on VM/CMS; Mike VanDerMeulen and John Mizel for helping us port the interface to the RISC System/6000; Ray Trimble, Michael Wheatley, Nancy Wheeler, and David Liebtag for helping us through the □NA of APL2; Elbert Hu for including APL2/X with TCP/IP Version 2 for VM; and the many people that answered our queries on the electronic forums and provided us with good feedback during the development.

* Trademark or registered trademark of International Business Machines Corporation.

** Trademark or registered trademark of Massachusetts Institute of Technology, UNIX System Laboratories, Inc., Microsoft Corporation, or Open Software Foundation, Inc.

Appendix A: HelloWorld listing

```
[0]    HelloWorld;□IO;dp;w;gc;s;e;k;rw;bp;wp;m
                 ;hello;hi;done;None;hp;hints;rc
                 ;nl;x;ep
[1]    ⍝ Sample X program, based on helloworld.c
[2]    ⍝ from Oliver Jones:
[3]    ⍝ Introduction to the X Window System
[4]    ⍝ Prentice-Hall, 1989;  ISBN 0-13-499997-4
[5]    □IO←0
[6]
[7]    ⍝ Define some constant text-strings
[8]    hello←'Hello, World.'
[9]    ⍝ The exclamation point makes hi ugly:
[10]   hi←'Hi',('A'=□AF 65)⊃□AF 90 33
[11]
[12]   ⍝ Initialization
[13]   →(0=dp←X 'XOpenDisplay' '')↓lopen
[14]   □←'XOpenDisplay failed ...'
[15]   □←'... HelloWorld aborted'
[16]   →lexit
[17]   lopen:
[18]
[19]   ⍝ Default pixel values
[20]   s←X 'XDefaultScreen' dp
[21]   bp←X 'XBlackPixel' dp s
[22]   wp←X 'XWhitePixel' dp s
[23]
[24]   ⍝ Define an X constant
[25]   None←0
[26]
[27]   ⍝ Prepare to set window position and size
[28]   (rc nl)←'H' axGetFF 'XSizeHints'
[29]   m←+/PPosition PSize
[30]
[31]   ⍝ Build an XSizeHints structure instance
[32]   hp←X 'New' 'XSizeHints'
[33]   hints←X 'Get' 'XSizeHints' hp
[34]   hints[H_flags H_x H_y]←m 200 300
[35]   hints[H_width H_height]←350 250
[36]   X 'Put' 'XSizeHints' hp hints
[37]
[38]   ⍝ Window creation
[39]   rw←X 'XDefaultRootWindow' dp
[40]   x←hints[1 2 3 4],5 bp wp
[41]   w←X 'XCreateSimpleWindow' dp rw,x
[42]   x←hello hello None('A' 'test')2 hp
[43]   X 'XSetStandardProperties' dp w,x
[44]   X 'SFree' 'XSizeHints' hp
[45]
[46]   ⍝ Create a Graphics Context
[47]   gc←X 'XCreateGC' dp w 0 0
[48]   X 'XSetBackground' dp gc bp
[49]   X 'XSetForeground' dp gc wp
[50]
[51]   ⍝ Window mapping
[52]   X 'XMapRaised' dp w
[53]
[54]   ⍝ Input event selection
[55]   m←'ButtonPressMask' 'KeyPressMask'
[56]   m←m,⊂'ExposureMask'
[57]   (rc m)←m axGetFF1 'XEvent'
[58]   X 'XSelectInput' dp w(+/m)
[59]   ep←X 'XGetEventBuffer'
[60]
[61]   ⍝ Get some more constants
[62]   m←'KeyPress' 'ButtonPress'
[63]   m←m, 'Expose' 'MappingNotify'
[64]   (rc m)←m axGetFF1 'XEvent'
[65]   ⍝ ... and some event structure layouts
[66]   nl←nl,1⊃'K_' axGetFF 'XKeyEvent'
[67]   nl←nl,1⊃'B_' axGetFF 'XButtonEvent'
[68]   nl←nl,1⊃'E_' axGetFF 'XExposeEvent'
[69]
[70]   ⍝ Main event-reading loop
[71]   done←0
[72]   levent:→(done=0)↓lend
[73]
[74]   ⍝ Read and process the next event
[75]   x←lKeyPress lButtonPress
[76]   x←x,lExpose lMappingNotify
[77]   →(m=K_type⊃e←X 'XNextEvent' dp)/x
[78]
[79]   lExpose: ⍝ Repaint window on expose events
[80]   →e[E_count]↑levent    ⍝ Count > 0 ?
[81]   x←e[E_display E_window],gc,50 50
[82]   X(⊂'XDrawImageString'),x,hello(⍴hello)
[83]   →levent
[84]
[85]   lButtonPress: ⍝ Process mouse-button presses
[86]   x←e[B_display B_window],gc,e[B_x B_y]
[87]   X(⊂'XDrawImageString'),x,hi(⍴hi)
[88]   →levent
[89]
[90]   lKeyPress: ⍝ Process keyboard input
[91]   k←∈1⊃X 'XLookupString' ep
[92]   →(done←(↑k)∈'qQ')↑levent
[93]   x←e[K_display K_window],gc,e[K_x K_y]
[94]   X(⊂'XDrawImageString'),x,k(⍴k)
[95]   →levent
[96]
[97]   lMappingNotify: ⍝ Reset keyboard
[98]   X 'XRefreshKeyboardMapping' e
[99]   →levent
[100]
[101]  /end: ⍝ Termination
[102]  X 'XFreeGC' dp gc
[103]  X 'XDestroyWindow' dp w
[104]  X 'XCloseDisplay' dp
[105]  nl←□EX'nl
[106]  lexit:
       ∇  1991-4-16 18.43.0 (GMT-4)
```

Cited references

1. J. A. Brown, S. Pakin, and R. P. Polivka, *APL2 at a Glance*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).
2. O. Jones, *Introduction to the X Window System*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
3. D. Comer, *Internetworking with TCP/IP*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).
4. *IBM Transmission Control Program/Internet Protocol Version 2 for VM: Programmer's Reference Manual*, Appendix A, SC31-6084, IBM Corporation (1990); Program No. 5735 FAL; available through IBM branch offices.

5. *IBM Transmission Control Program/Internet Protocol Version 2 for MVS: Programmer's Reference Manual*, SC31-6087, IBM Corporation (1991); Program No. 5735 HAL; available through IBM branch offices.
6. *X3270—AIX X Windows 3270 Emulator User's Guide*, SC23-0579-0, IBM Corporation (1991); available through IBM branch offices.
7. J. A. Pierce and R. O. Reynolds, *The X Window System in the S/370 Environment*, G325-4100-0, IBM Corporation (1991); available through IBM branch offices.
8. *APL2 Programming: Processor Interface Reference*, SH20-9234-0, IBM Corporation (1987), pp. 15–23; available through IBM branch offices.
9. *APL2 Programming: APL2 for the IBM PC, User's Guide, Version 1.02*, SC33-0600-2, IBM Corporation (1990), pp. 436–438; available through IBM branch offices.
10. R. W. Scheifler and J. Gettys with J. Flowers, R. Newman, and D. Rosenthal, *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFS*, Digital Press, Bedford, MA (1990).
11. *AIX Calls and Subroutine Reference for RISC System/6000, Volume 4: User Interface*, SC23-2198, IBM Corporation (1990); available through IBM branch offices.
12. *APL2 Programming: System Services Reference*, Chapter 23, SH20-9218, IBM Corporation (1990), pp. 238–239; available through IBM branch offices.
13. *Ibid.*, p. 237.
14. *Ibid.*, p. 243.
15. See Reference 2, Chapter 2.
16. *AIX APL2/6000 User's Guide*, SC23-3051-0, IBM Corporation (1991), pp. 305–314; available through IBM branch offices.
17. *An Interface Between APL2 and the X Window System*, IBM licensed material provided with TCP/IP Version 2 for VM (Program No. 5735-FAL), pp. 5–14; available through IBM branch offices.
18. See Reference 16, pp. 209–216.
19. See Reference 17.

Accepted for publication June 10, 1991.

John R. Jensen *IBM Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Mr. Jensen is a scientific staff member at the Cambridge Scientific Center. He joined IBM Denmark in 1978 as a systems engineer. In 1982, he worked at the IBM Canada Computing Centre in Vancouver, British Columbia, and from 1983 to 1988 was at the Dallas Development Laboratory in Texas, working on the IC/1 and OfficeVision products. He became a member of the Cambridge Scientific Center staff in 1988. His current areas of interest include user interface design, application prototyping, programming environments, and compilers. Mr. Jensen received an M.Sc. degree in electrical engineering from the Technical University of Copenhagen, Denmark, in 1978 and an M.B.A. in accounting from the Copenhagen School of Economics in 1981. He is a member of the ACM and the IEEE Computer Society.

Kirk A. Beaty *IBM Cambridge Scientific Center, 101 Main Street, Cambridge, Massachusetts 02142.* Mr. Beaty is a scientific staff member at the Cambridge Scientific Center. He joined IBM at Sterling Forest, New York, in 1981 as a systems programmer. Furthering his experience at Sterling Forest, from 1983 to 1987 he became involved in telecommunications, including the technical software leadership role in the creation of IBM's centrally managed internal VNET backbone network. He has been a member of the Cambridge Scientific Center since 1987. Mr. Beaty received a B.S. with honors in mathematics/computer science (while minoring in business administration) in 1981 from Manchester College, North Manchester, Indiana. He is a graduate of the IBM Systems Research Institute and has recently completed a Certificate of Advanced Study in software engineering at Harvard University.

Reprint Order No. G321-5447.

The APL IL Interpreter Generator

by M. Alfonseca
D. Selby
R. Wilks

The objective of the APL IL Interpreter Generator is to solve the problem of creating APL interpreters for different machines at a minimum cost. The objective has been accomplished by writing an APL interpreter in a specially designed programming language (IL) that has very low semantics but high-level syntax. The interpreter is translated to each target machine language by easily built compilers that produce high-performance code. The paper describes IL, the APL interpreters written in IL, and the final systems generated for seven different target machines and operating systems. Some of these systems have been generated in an extremely short time.

Among the many languages used to write programs, APL and its successor, APL2, are very powerful. They support highly structured data of several different internal types and recognize a large number of primitive functions and operators, some of which (for example, **execute**, ⍎) are extremely complicated for some arguments. The existence of these primitives makes it very difficult for APL to be compiled (except for subsets of the language or through the inclusion of an interpreter in the machine code). Thus full APL and APL2 systems have to be interpretive. These interpreters are very large programs, consisting of tens of thousands of instructions.

Since interpreted programs normally run at least an order of magnitude slower than their compiled equivalents, programs written in APL or APL2 start with a speed handicap as compared to programs written in, say, C. However, the designers of APL and APL2 and the implementers of the interpreters have tried to reduce this effect in two different ways:

- By extending the language with ever more powerful primitives. In a single stroke, these perform complex operations that, in other languages, would require complicated algorithms. In this way, the time for interpretation is minimized with respect to the time for execution. The fact that most APL primitives apply to entire arrays also helps in this direction.
- By programming the interpreters in very low-level languages that make the best possible use of the resources of the machine or the operating system.

As a result, APL and APL2 interpreters were usually written in assembly languages, with the consequent loss of portability. It has been estimated several times that, done in this way, the full development of an APL system for a new machine requires a total of about 30 person-years.

The APL IL Interpreter Generator started as a project in the IBM Madrid Scientific Center in 1977. The objective of this project was to solve the problem of obtaining APL interpreters for different machines, at a minimum cost. The solution was to write an APL interpreter in a programming language, specially

designed for the purpose, that has very low semantics but high-level syntax. This interpreter is translated to each target machine by appropriate, easily built compilers that produce high-performance code.

In the past 14 years, the programming language called the Madrid Scientific Center Intermediate Language (IL) has reached its third version; it has been essentially stable since 1980. The first section of the paper describes the language design decisions, which in many cases are curiously parallel to those made in the design of the C language, although there are important differences. The second section of the paper describes the different interpreters that have been written in IL since 1980. Finally, the last section describes the procedure used to generate an APL system for a given target environment (a machine and an operating system).

The Intermediate Language

The Madrid Scientific Center Intermediate Language (IL) was designed in the late 1970s, according to the following criteria: on the one hand, a high-level syntax was desirable to assure portability between different machines and operating systems; on the other hand, very low-level semantics would make it possible to obtain highly optimized code with very simple, easy-to-build compilers.

The procedure that was followed to design the IL instructions was to select the most common operations in the assembly languages of different IBM machines and to represent them with a high-level syntax. In this way, compilation of IL instructions into assembly language usually becomes a one-to-one translation between one IL symbol and one assembly instruction.

Even control instructions were subject to this procedure. Since the only control instruction in assembly languages is usually the *branch on condition*, this instruction is the only one that was implemented in IL, although it received a high-level syntax in the following way:

```
→label IF condition
```

Optimization, in this kind of intermediate language, is not a question to be solved by the compilers, which we want to build as quickly as possible, but by the IL programmers who write the APL interpreter. Remember that this job should be done only once, although there may be as many compilers as there are different target machines.

The only assumption about the machine in which IL may eventually be implemented is that its memory is considered to be a vector of fixed but undefined size (eight bits or more per byte; two, four, or eight bytes per word). Memory units should be consecutively numbered.

The four elements of IL are now described.

Constants. Constants can be numeric or literal. In actual fact, a literal constant can also be considered as numeric and operated on accordingly. This means that an expression such as

```
'A'+1
```

is valid and (assuming ordinality in the character set) is equivalent to constant

```
'B'
```

The C language manages character constants in the same way.

ASCII (American Standard Code for Information Interchange) or EBCDIC (extended binary-coded decimal interchange code) can be selected as the internal representation of the literal constants. In the case of the APL IL interpreters, ASCII has been chosen.

Numeric constants can be either integer or floating point. Floating-point constants, such as `2.0`, are distinguished by the presence of the period from integer constants, such as `2`.

Identifiers. Identifiers are names that begin with a letter other than Q (which is reserved) and continue with any (possibly empty) combination of letters and figures. The maximum number of characters in an identifier is five.

What an identifier represents is controlled by its first letter, according to Table 1.

A full-word variable has an implementation-dependent length. Depending on the machine (in a 16-bit system, for instance), a full-word variable can be the same as a two-byte variable. This type is, to a certain extent, similar to the **int** type in the C language, but IL does not distinguish full-word in-

Table 1 Identifiers and their definitions

Identifier	Representation
O,R,T,U,V,W	A variable whose value is a vector of one-byte integers or literals.
I,J,K,L,M,N	A variable whose value is a vector of two-byte integers.
A,B,C,D,G,H,P	A variable whose value is a vector of full-word integers or pointers.
F	A variable whose value is a vector of floating-point values.
E	An internal label in a program.
S	A public label in a program.
X,Y,Z	A named constant.

tegers from pointers. Assembly languages do not usually make this distinction either.

The only data structure supported is the vector (a succession of values at consecutive locations). Higher structures (such as matrices) are not a part of IL, as they are not a part of assembly languages. A scalar is considered to be the same as a vector of one element.

Declarations. In an IL program, declaration instructions are located at the beginning and clearly separated from executable instructions. Every variable used by a program must be declared, either by assigning initial values to it, or by defining an equivalence.

Initial values are assigned by means of instructions such as the following:

```
A←1 3 5 7
B←10ρ0
W←'ABC'
```

The first instruction defines A as a vector of four full words with initial values of one, three, five, and seven. The second defines B as a vector of ten full words with initial values of zero. The third defines W as a vector of three bytes with initial values equal to the ASCII representation of letters A, B, and C.

Equivalences are very powerful and have different forms, such as:

```
C=A[2]
V=8ρF
C1=3ρP1(4)
```

The first instruction defines variable C to have the same address as the third element of vector A (zero origin is used). Both A and C are full-word objects by virtue of their initial letter.

The second instruction defines V as a vector of eight bytes, sharing the address of floating-point variable F. This means that V is the vector of the bytes that make up the floating-point value of F, assuming that floating-point values are represented in eight bytes.

The third instruction defines $C1$ as a vector of three full words whose address is the current value of pointer $P1$ plus four. Of course, if the value of $P1$ changes, the address of C will change accordingly.

Pointers are extremely useful in IL programs, just as they are in C. However, there is no restriction on the number of equivalences that may be defined to a pointer at the same time. For instance, the following declarations

```
A1=4ρP(0)
I1=P(0)
V1=P(0)
```

are all valid and define three variables that share the same address (the value of pointer P), but have a different type. $A1$ is a pointer or full-word integer vector of four elements. $I1$ is a two-byte scalar, and $V1$ is a one-byte scalar.

Executable instructions. Executable IL statements are analyzed and executed from right to left. Functions are executed without any precedence rules in the order in which they are found. Parentheses are not allowed. The main IL executable instructions are of two different types: assignment instructions and execution control instructions.

Assignment instructions may take four different forms, according to the following syntax:

```
variable ← expression
variable Δ expression
variable ∇ expression
pointer_variable → address expression
```

where the first form corresponds to normal assignment, the second increments the value of the variable by the right-hand expression, the third decrements that value in the same way, and the fourth, only applicable to pointers, assigns to the variable the address of the expression on the right side.

Execution control statements have three different forms:

```
→label
→label IF condition
→label_list OF index
```

where the first form corresponds to the unconditional transfer, the second to the conditional transfer, and the third to a computed go-to instruction.

The operations that can be a part of an expression are the typical ones usually encountered in most machine languages, such as the following: addition (+), subtraction (−), multiplication (×), division (÷), residue (\|), bit shift to the left (↑), bit shift to the right (↓), bit-to-bit logical operations that include not (~), and (∧), inclusive or (∨), and exclusive or (∘), absolute value (\| in monadic form), and an operation to compute the integer part of a floating-point number (~ in monadic form).

The following is an example of an executable instruction in IL:

```
P1←AREF+ZEI4↑4+1↑DREFI
```

This instruction computes the value of pointer *P1* in the following way: The value of variable *DREFI* is multiplied by two (a shift to the left of one position is equivalent to a multiplication by two); then, four is added to the preceding result. Next, the new result is shifted to the left by as many positions as the value of constant *ZEI4* (which depends on the target machine). Then the value of variable *AREF* is added, and finally, the result is assigned to pointer *P1*.

Another kind of executable instruction is the subroutine call. Its syntax is very simple, just the name of the subroutine. No parameters can be passed explicitly. All of them must be passed through common memory, or by means of a set of special pointer variables, the values of which are automatically restored before returning to the calling routine. These variables fulfill the role of the machine registers, and in fact, in several of our implementations they are registers, but this is not necessarily so.

Language tradeoffs. A question that could be discussed is whether IL has any advantages over C for the implementation of machine-independent software. This question is really after the fact since IL was designed in 1977, at a time when C was in its infancy and far from being as widespread as it is now. However, in our opinion, IL is superior to C in its memory management capabilities, which are much nearer to the machine language level, and also in its ability to define multiple pointer-based structures that can overlap freely and move around without any restrictions.

In contrast, C has better type-constraint capabilities that provide the programmer with mechanisms to detect certain errors at compile time, which IL compilers do not have. However, we did not find the lack of these capabilities frustrating in our development of APL interpreters.

Finally, IL, being a less complicated language, can be translated by very simple compilers. This point was important in our development procedure, which is described in the last section of this paper.

The APL IL interpreters

IL has been used for the development of several different interpreters. In the time from 1978 to 1982, an APL interpreter was built at about the same level of the language as the one implemented in the VS APL product. This interpreter was especially applicable to small machines with reduced data spaces in memory, and to increase the amount of workspace available to the user, we introduced the concept of an elastic workspace. This interpreter was compiled into the System/370* (which we used as our test machine), the Series/1*, and the IBM Personal Computer.

The Series/1 computers had an important limitation: the memory data space used by one application was restricted to 64K bytes. To increase it, we implemented the elastic workspace as a disk extension of the workspace. APL objects directly accessible to the user (in the active workspace) could also reside on disk and would be copied into the main memory only when they were needed.

When the IBM Personal Computer was announced in 1981, we decided to translate our interpreter to

this machine under the Personal Computer Disk Operating System (PC DOS, or DOS). In this case, there was the same limitation in the fact that the use of segment registers made only 64K bytes directly available. But these machines have greater flexibility in comparison to the Series/1, since the contents of the segment registers can be changed by the program. This flexibility made it possible for us to implement the elastic workspace extension in main memory, which made it much faster and more efficient.

The workspace was divided into two different sections. In the first section, with a length of 64K bytes, all the objects were directly accessible to the programs. This section included the APL symbol table, the APL execution stack, and many APL objects, all of them smaller than 32K bytes.

The second section (the elastic workspace) contained APL objects larger than 32K bytes and (possibly) smaller APL objects that did not fit in the directly available workspace at a given time and were not currently needed. Depending on the amount of space available (limited in DOS to 640K bytes but possibly reduced by the actual memory of the machine and the loading of the operating system extensions), the elastic workspace could be automatically reduced to zero.

This organization made it possible to build the IL compiler for the IBM Personal Computer in such a way that the compiler could assume that all of the objects are directly accessible and forget about segment registers. The only module not complying with this restriction was the handler of the elastic workspace, which was written directly in assembly language.

However, the indicated memory organization had an important disadvantage: many of the basic APL structures, such as the symbol table and the execution stack, could not increase further than 32K bytes, and users soon found that this was a strict limitation. Therefore, during 1983–85, we developed a new APL interpreter with a more general workspace management, specially adapted for 16-bit addressed segmented microprocessors (such as the i8086). This interpreter, which from the language point of view was still at the VS APL level, was compiled into the System/370 (which we always use as the test machine) and also into the IBM Personal Computer (under DOS) and the IBM Japanese Personal Computer and JX PC (under Japanese DOS) as a result of a joint project between the IBM Madrid and Tokyo Scientific Centers.

The elastic workspace concept was abandoned, or (as it may be preferred) extended to the whole workspace. In actual fact, what happened is that this system incorporated a single workspace area containing all of the APL objects, including the symbol table and the execution stack, regardless of their sizes. The lower part of the workspace, however, always directly accessible through the base segment registers, includes all of the interpreter data and work areas plus four "operand areas."

This organization made it possible to build the IL compiler for the IBM Personal Computer so that the compiler could assume all of the objects are directly accessible.

An operand area is a section of the workspace located in the lower 64K bytes of the total workspace, where the system can copy APL objects, either completely or partially. A set of special subroutines manages the transfer of the data from the operand areas to the workspace proper and vice versa. The remainder of the interpreter works only with the operand areas and can thus forget about the segment registers. Only a few modules in the whole interpreter (less than 10 percent) must work directly on the workspace, and thus they must be hand-modified in assembly language to introduce the required modifications to the segment registers.

In 1985 we started a joint project between the Madrid and United Kingdom Scientific Centers to build an APL2 interpreter written in IL. This interpreter has been compiled, as usual, into the System/370, and also to the following target machines and operating systems: the IBM Personal Computer (IBM PC), Personal Computer AT*, and Personal System/2* (under DOS and Operating System/2*, or OS/2*), the IBM Japanese Personal Computer (under Japanese DOS), the Intel 80386**-based machines in 32-bit addressing mode (under DOS), the

Table 2 Previously available interpreters

1. IBM Personal Computer APL, version 1.0, Program Number 6024077, 1983.
2. IBM Personal Computer APL, version 2.0, Program Number 6391329, 1985.
3. 5550 NiHonGo (Japanese) APL, version 1.0, Program Number 5600-JPL, 1984, developed in collaboration with the Tokyo Scientific Center.
4. 5550 NiHonGo (Japanese) APL, version 2.0, Program Number 5600-JPN, 1985, developed in collaboration with the Tokyo Scientific Center.
5. JX NiHonGo (Japanese) APL, Program Number 5601-JPL, 1985, developed in collaboration with the Tokyo Scientific Center.
6. APL2 for the IBM Personal Computer, version 1.0, Program Number 5799-PGG (PRPQ RJ0411, Part No. 6242936), 1988.
7. APL2 for the IBM Personal Computer, version 1.0E, Program Number 5604-260 (Part No. 38F1753), and Program Number 5775-RCA (Part No. 38F1754), 1988.
8. APL2 for the IBM RISC System/6000, Program Number 5765-012, 1991, developed in collaboration with the APL2/6000 Development Group from the IBM Kingston Laboratory.

IBM 6150 RT PC* (under Advanced Interactive Executive*, or AIX*), and the IBM RISC System/6000* (under AIX).

There are two versions of this interpreter. The first one, used to generate the PC-like 16-bit systems, still uses the memory management described for the second APL interpreter. However, in the second APL2 interpreter used to generate the 32-bit systems, where memory management is not a problem, some of the modules have been replaced by others that work directly on the workspace, skipping the copy to the operand areas, to improve performance.

An additional improvement in the APL2 interpreters is the presence of a reference table, functionally intermediate between the symbol table and the actual APL objects. This improvement means that most of the time the interpreter may refer to a given object by its reference number, regardless of the actual position where the object is located in the workspace. There are several important consequences of this organization that are now described.

On the one hand, a given APL2 piece of data may be pointed to by more than one APL object. Since APL2 supports general arrays, this capability is important to prevent memory duplication. The reference table keeps information that indicates whether an object is multipointed, which will be used in case of modification to decide whether the value should be copied somewhere else before the changes are performed.

On the other hand, garbage collection is much simplified and made extremely fast. This has always been the case with APL, but it is even more dramatic now that extremely large workspace sizes can be attained. With our 32-bit interpreter, workspace sizes can reach many megabytes, but even so, garbage collection never takes longer than a few seconds. This speed contrasts with other interpretive languages, such as LISP and Smalltalk, where garbage collection was traditionally a very expensive procedure, sometimes taking several minutes to complete.

Table 2 lists some of the previous interpreters that have become international IBM products.

Generating an APL interpreter

The procedure to generate an APL2 system for each environment (machine and operating system) can be summarized as follows. First, a compiler that translates IL code into the target machine code is built. Next, the APL2 IL interpreter is compiled into the target machine code. This produces an incomplete system, with a few loose ends (subroutines) that depend on the operating system and that have not been written in IL. These subroutines are then written, usually in assembly language, and added to the compiled interpreter. Finally, some auxiliary processors are written to perform special I/O operations.

This procedure has proved its usefulness in the fast and effective generation of APL2 interpreters for different machines. The outstanding example was the i80386 interpreter, where we could get rid of the fourth step (since we took care that all auxiliary processors written for the IBM Personal Computer and Personal System/2 [PS/2*] interpreter would be compatible). The total effort required to execute the other three steps and produce and debug a full APL2 system for these machines was 13 person-weeks. The system was announced and shipped just six months after the work started.

Another outstanding example was the porting of the APL2 IL interpreter to the IBM RISC System/6000 machine under the AIX operating system. It was done in about ten person-weeks by two people who did not have previous knowledge of either IL or the RISC System/6000 machine code.

IL compilers. The IL compilers are usually written in APL or APL2, which makes them very easy to adapt to new target machines. They are somewhat slow since they are being interpreted, but this is not a problem since, in principle, they need only be executed once.

When an APL system must be generated, it is usually not necessary to build a full IL compiler.

When an APL system must be generated for a new machine or operating system, it is usually not necessary to build a full IL compiler. Since the source language is the same, the lexical and syntax analysis sections of any of the preceding compilers are automatically usable. Only the code generator section must be rewritten, and even there, many subprograms and program structures can be reused.

The exception is the IL-to-System/370 compiler, since we are using the System/370 as a test machine and many changes and trials are performed on it. Therefore, the IL-to-System/370 compiler was written in IL and is much faster than all of the other IL compilers.

At this point, we have IL compilers available for the System/370, the Series/1, the i8086 and i80286 machines (which include the IBM Personal Computer, the PS/2 Models 25, 30, 50, and 60 and the Japanese IBM PC), the i80386 machines (such as PS/2 Models 70, 80, 90, and 55SX), the IBM 6150, and the IBM RISC System/6000. The last three compilers are written in APL2.

Operating-system-dependent code. Operating-system-dependent code performs those functions that depend closely upon the operating system and are not easily made machine-independent. They include system initiation and disconnection, machine check recovery, console I/O, sequential file I/O, and the timer routines. This code, as compared to the size of the APL interpreter, amounts to about 5 percent of the total code.

In the case of the Series/1, we also implemented a time-sharing system able to support the simultaneous use of the machine by several users. This system was written directly in assembly language, and its presence increased the amount of machine-dependent code to about 10 percent of the total code of the system.

Auxiliary processors. Auxiliary processors are written for the management of different peripherals and specialized computations. They perform functions such as loading and execution of external programs, printer interface, operating system interface, full screen management, data file processing, communications, graphics, music generation, special device drivers, and logic programming.

Not all of these auxiliary processors are available for all of our target machines. Some of them are written in IL, some in C, and some in assembly language. A few of them (such as the special device drivers) are not only machine- and operating-system-dependent, but also hardware-attachment-dependent. It makes no sense to develop them in a high-level language, since assembly language always provides the maximum efficiency.

Conclusion

The APL IL Interpreter Generator has proved its usefulness in generating APL and APL2 interpreters with a considerable reduction of the total product cycle. It has been used to generate nine IBM products: the eight APL and APL2 systems listed previously, plus an educational product announced by IBM Japan, called LETSMATH, that includes the interpreter without the user being aware of it. Several additional systems, restricted for IBM internal use, have also been generated in the same way.

* Trademark or registered trademark of International Business Machines Corporation.

** Trademark or registered trademark of Intel Corporation.

Cited references

1. *APL Language*, GC26-3847, IBM Corporation (1975); available through IBM branch offices.
2. *APL2 Programming: Language Reference*, SH20-9227, IBM Corporation (1987); available through IBM branch offices.
3. M. Alfonseca and M. L. Tavera, "A Machine-Independent APL Interpreter," *IBM Journal of Research and Development* **22**, No. 4, 413–421 (July 1978).
4. M. L. Tavera, M. Alfonseca, and J. Rojas, "An APL System for the IBM Personal Computer," *IBM Systems Journal* **24**, No. 1, 61–70 (1985).
5. M. Alfonseca and D. Selby, "APL2 and PS/2: The Language, the Systems, the Peripherals," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 4, 1–5, ACM, New York (1989).

Accepted for publication June 21, 1991.

Manuel Alfonseca *IBM Software Technology Laboratory, Paseo de la Castellana, 4, 28046 Madrid, Spain.*

Dr. Alfonseca is a Senior Technical Staff Member in the IBM Software Technology Laboratory. He has worked in IBM since 1972, having been previously a member of the IBM Madrid Scientific Center. He has participated in a number of projects related to the development of APL interpreters, continuous simulation, artificial intelligence, and object-oriented programming. Eleven international IBM products have been announced as a result of his work. Dr. Alfonseca received electronics engineering and Ph.D. degrees from Madrid Polytechnical University in 1970 and 1971, and the Computer Science Licenciatura in 1972. He is a professor in the Faculty of Computer Science in Madrid. He is the author of several books and was given the National Graduation Award in 1971 and two IBM Outstanding Technical Achievement Awards in 1983 and 1985. He has also been awarded as a writer of children's and juvenile literature.

David Selby *IBM United Kingdom Scientific Centre, Athelstan House, St. Clement Street, Winchester, Hants, SO23 9DR, England.*

Mr. Selby joined IBM at the Havant manufacturing location in 1977, where he worked on many APL projects in the capacity of analyst programmer and later as a microcode engineer for the 4700 Finance Industry System. In 1985 he joined the Scientific Center at Winchester in the Graphics Systems Research group as a scientist employed on workstations. Beginning in 1983, he collaborated with Dr. Alfonseca on APL/PC 2.0 with special emphasis on auxiliary processors and device support. The result of this work was used in APL/PC 2.0, Japanese PC APL 2.0, and the APL2/PC products. Mr. Selby is also responsible for the design of the extended memory driver of the 32-bit version of the APL2/PC interpreter and has worked as a technical consultant to the APL2/6000 project. While working for IBM, he has obtained an ONC in electrical engineering, and an HNC in computer science. He has also received an IBM Exceptional Achievement Award.

Ron Wilks *IBM United Kingdom Scientific Centre, Athelstan House, St. Clement Street, Winchester, Hants, SO23 9DR, England.*

Mr. Wilks joined IBM in 1973 and has used APL since his very first day with IBM. He started his career in a group developing diagnostics for small disk files. From there, he joined the IBM Hursley Information Systems (IS) Applications Support group to support APL and related products. While in IS, Mr. Wilks assisted with enhancements to the APL/PC Version 1 product culminating with the announcement of the APL/PC Version 2.0 product for which he received an IBM Exceptional Achievement Award. After IS, he joined the small group of APL2/PC developers to assist with APL2 for the IBM PC product and, more recently, the AIX APL2/6000 product.

Reprint Order No. G321-5448.

Parallel expression in the APL2 language

by R. G. Willhoft

This paper reports on an investigation of parallel expression and execution in the current APL2 language. The study covers a historical, theoretical, and empirical viewpoint. The parallel nature of APL is traced from its foundations in the Iverson notation to current problems in executing APL on parallel hardware. The paper discusses features of the APL language and its current implementations that limit taking advantage of parallel expressions. A survey of related topics from the work on APL compilers is also included. Each APL2 language construct is examined for potential parallel expression. The operations are grouped based on the possible parallelism exhibited by each operation, and the possible implementation of each group is discussed. Three APL2 applications are explored to determine the actual parallelism expressed in "real" APL2 code. These applications are chosen from distinct areas: graphics, database systems, and user interactive systems. The actual data passed as arguments to every operation are dynamically examined, and the information is collected for analysis. The data are summarized and results of the study are discussed.

In the last several years, APL has received attention as a language that can be used to express parallel algorithms. The primary interest has been in the ability of the language to express algorithms on vector or array arguments directly, eliminating the need for a programmer to convert them into sequential loops. The question to be addressed in this paper is: Given a powerful array language, how much parallelism is expressed implicitly? This study attempts to better understand the extent of parallel expression that is contained in typical APL2 applications.

The parallel nature of APL2 is investigated in two ways that make this paper unique from similar studies in the past. First, there is an emphasis on completeness. All APL2 primitives are examined for possible parallel execution. Next, there is an emphasis on gathering empirical information. This study measures real code to achieve a better understanding of the extent of parallel expression in "real" APL2 code.

Once the parallel nature of current APL2 is understood, this paper also answers two other questions: From a language viewpoint what items could be changed to increase the parallel expression in the language; and what lessons can be learned regarding the development of parallel interpreters for the current APL2 language?

The parallel nature of APL

APL: A parallel language. APL is a language that can be considered parallel since its very inception. Ken Iverson, in his original definition of A Programming Language,[1] defines a language that is at its very roots a parallel language. The *Iverson notation* (the name used to describe the notation in Iverson's book) was not intended to be implemented. However, APL and APL2 were developed directly from the concepts that he outlined.

The 25 years of APL history have been scattered with work that has attempted to extract and exploit

the power of the Iverson notation. Recently much of that work has been focused on using APL (or APL-like notation) on parallel machines. The advantage of APL for parallel applications was recognized as early as 1970 by Abrams:

> In general, APL programs contain less detail than corresponding programs in languages like ALGOL 60, FORTRAN, or PL/I.... While this aspect of APL often makes programs shorter and less intricate than, say, ALGOL programs, it also requires that an evaluator of APL be more complex than one for ALGOL, especially if such expressions are to be evaluated efficiently. On the other hand, a machine doing APL has greater freedom since its behavior is specified less explicitly. In effect, APL programs can be considered as descriptions of their results rather than as recipes for obtaining them.[2]

The following sections explore the history of APL as it relates to execution on parallel machines.

Types of parallel expression. Parallel expression can be classified in a number of ways. The terms *course grain* and *fine grain* have been used to distinguish the size of the tasks that are executed in parallel. *SIMD* (simple instruction stream, multiple data stream) and *MIMD* (multiple instruction stream, multiple data stream) concentrate on the nature of the instructions that are issued to perform the calculations, and *vector processor*, *array processor*, and *multiprocessor* tend to emphasize the difference in the machine architectures that are used for parallel execution. All of these terms interrelate and are often used interchangeably.

For the purpose of this work, four terms will be introduced that focus on the nature of the expression from which the parallelism is extracted. They are *data parallelism*, *algorithm parallelism*, *data-flow parallelism*, and *task parallelism*.

The first three types of parallel expression are implicit—parallelism is "implied" by the operation(s) specified instead of being explicitly stated by the programmer. Task parallelism is the one explicit parallel expression.

Data parallelism in APL. Data parallelism refers to the application of a single conceptual operation to a number of data items at the same time. Each of the operations is completely independent from the rest. Hillis has coined the term *data parallel* to distinguish the difference in parallelism that comes from simultaneous operations across large sets of data, rather than from multiple threads of control.[3] The key concept of this definition is the fact that the expression of parallelism comes from the specification of operations across sets of data.

Although Hillis connects the idea of threads of control to his definition, our use of the term will not make this connection. There are times that the execution of a single conceptual operation to a set of data items will require, or at least allow, independent and distinct algorithms to be run on the separate data items. Although the execution in this case is MIMD, instead of the SIMD implied in Hillis's definition, the expression of parallelism is still of the data parallel form.

The concept of arrays of data is not unique to APL. What sets APL apart is that arrays in APL are viewed as a unified whole, rather than a collection of individual data items.[4] This view is what made Iverson's work so powerful. Iverson also defined operations on arrays including element-wise application of functions, scalar extension, selection, reduction, and permutation operations. The power of these concepts has been recognized in the work on new parallel languages[5] and in the work to include parallelism in existing languages, such as FORTRAN.[6]

Brenner[7] outlines some of the considerations and advantages of implementing APL on an array processor similar to the Connection Machine.[8] Brenner recognized the potential of execution of scalar functions, scan, and reduction on a parallel processor. Brenner also gives a thumbnail sketch of how some other APL operations might be executed in parallel. He outlines methods for **compress**, **expand**, **grade up**, **reshape**, **rotate**, **take**, **drop**, **index of**, **member**, and **inner product**. Although this is an impressive list, it is only a small part of the operations that can be done in parallel, as will be shown in this paper.

The parallel execution of APL has not only been shown theoretically, but also has been implemented in several machines. The Analogic APL Machine, introduced in 1980, used the APL language to drive a vector processor. As Delo points out, "One important achievement of the project is running software ... that had been written in a standard programming language to run on a conventional

computer."[9] Even today this is an achievement that has been matched by few other parallel computer projects.

While the APL machine was specially designed for APL execution, most parallel hardware is not designed with APL in mind. However, APL seems well positioned to take advantage of the new hardware. For example, the IBM 3090* Vector Facility is a high-performance pipeline processor designed to significantly improve vector performance.[10] APL2 was one of the first languages to use the Vector Facility for the processing of vector (array) data. The close match between the expressiveness of APL2 and the processing of the IBM Vector Facility has led Brown to conclude "... in some senses, the IBM Vector Facility is a machine designed for executing APL."[11]

Algorithm parallelism in APL. Algorithm parallelism refers to operations that can exploit the relationships of the data items to allow execution in parallel. This is in contrast to the assumption of independence among the items in data parallelism. In this form of parallelism, it is the algorithm that is parallel in nature. The data must be viewed as one item.

Examples are sorts, FFTs (Fast Fourier Transforms), matrix inversions, and similar operations. In each of these cases there are suboperations that can be executed in parallel, but these operations must be coordinated and supervised by an overall plan.

Although this type of parallel expression can clearly be replaced by algorithms written using the other parallel expression methods, the power of the expressiveness is lost. The advantage of capturing algorithm parallelism at the language level is that it allows for different architectures to execute the operation as is best suited for the machine.

Data-flow parallelism in APL. Data-flow parallelism results from the flow of results of one operation to arguments of the next operation. Since often there are multiple arguments to a given operation, each of those arguments can be calculated in parallel. To exploit data-flow parallelism it is necessary to calculate the data dependence (both argument and result) of each calculation. Then the order of calculation can be generated and is usually represented graphically. This directed graph shows the operations that can be executed in parallel.

This type of parallelism is by far the most difficult for the programmer to detect and exploit using explicit parallel expression. And although it is difficult for the system to detect this parallelism, the benefits of doing so are well worth the investment.

Most of the work that has been done in the area of data flow in APL has been in three areas. The first is work that is being done on developing an APL compiler.[12,13] Clearly, data flow is necessary to understand the manipulation of data in APL so that it can be compiled. The second area of work is in the area of functional languages. Backus[14] understood the potential that APL had as a functional language. Many have attempted to exploit this potential, usually with the goal of being able to create a parallel language based on functional constructs.[15,16] Finally, there are some who have looked at data flow solely as a method of execution within the APL language.[2,17,18]

In this section some of the methods and results of the work in all three areas are presented. The goal is to present the relationships between the work and some common ideas.

Abrams[2] and Wakshull[17] both explored the area of lazy evaluation. In this form of evaluation, values for arguments are not calculated until they are needed by the function that references them. Abrams used this idea to eliminate calculation on data that were later to be discarded, a concept he called "drag-along." Wakshull, while not discussing the benefits, gives a method by which an entire line of code can be executed using only data-flow principles.

Both Wakshull[17] and Ching[13] discuss the concept that both the left and right arguments to a function can be calculated at the same time. They formalize this concept by showing how a single dyadic function call can be placed within a pair of PARBEGIN and PAREND statements.

Budd[12] shows the power of constructing a complete data-flow graph. By doing so he is able to make statements about the rank, shape, and type of data variables. Although this benefit is connected with the problems of compiling APL, the technique is useful for discovering a number of properties of APL code without actually executing the code. For example, this type of analysis would be useful in determining interference between the assignments of two functions.

Task parallelism. Task parallelism expresses parallelism as separate tasks that are started and stopped by the application. These tasks run concurrently and may or may not communicate and synchronize with each other. All other forms of parallel expression can be broken down into task parallelism. The implicit parallel expressions already discussed are methods of hiding these operations from the user of the language, and therefore freeing the user to concentrate on the expression, not the control, of parallelism.

Task parallelism concentrates on the starting, stopping, synchronization, and communication between processes (tasks) at a level at which the user retains control over these operations. Task parallelism is exhibited in APL2 in the area of shared variables.

Shared variables, and the concept of auxiliary processors, are the oldest parallel facilities in APL. The auxiliary processor in APL can be a process running in parallel with the current workspace evaluation. The processing in the auxiliary processor is asynchronous to the workspace processing. The synchronization of the workspace with a given auxiliary processor is done with the shared variable. The shared variable is also used to pass commands to the auxiliary processor and to receive results from that unit.

APL2 has expanded the power and use of shared variables in several ways. Most importantly APL2 now allows variables to be shared between individual APL2 workspaces. In addition, several new shared variable system functions have been introduced that allow for more flexible methods of polling and using the shared variables. It has been noted by Gerth[19] that shared variables allow parallel structures without adopting artificial constructs in the language.

Hindrances to parallelism. There are some hindrances to parallelism in APL. These items must either be eliminated from the language or their effects must be minimized.

Assignments and side effects. One of the major problems in trying to execute code in parallel is that side effects may be produced. A side effect is any change in the state of the machine during the execution of a function that can be observed outside the function. Typical examples are assignments, I/O, and implicit results (such as the change to $\square RL$ made during the **roll** and **deal** functions). Side effects hinder parallelism because the total behavior of the program must create the same side effects in the same order to be a proper parallel implementation. Tu and Perlis[16] eliminate assignment in their functional language based on APL.

Dynamic binding. Dynamic binding causes the names in APL programs to be bound to values based on the environment in which the function is called. Dynamic binding makes it difficult to determine, before actual execution, many of the particulars of a program's activities. This complicates the areas of determining parallelism and avoiding interference. The alternative to dynamic binding is static binding. Static or lexical binding causes the values to be bound to the names based on the environment in which the object is defined. This solves many of the problems of program analysis and is therefore required by much of the data-flow work.[12,15,16]

Branching. The danger of GOTOs (branches in APL) have long been known by programmers. Specifically, in the area of parallel execution, branching makes it difficult to determine the exact execution of a program. At least two methods have been presented to deal with this problem. Some simply do not allow branching.[16] Others allow branching but only evaluate parallelism inside basic blocks (the areas between branches).[13]

Lack of declarations. Finally, the lack of declarations in APL deprives the interpreter (or compiler) of knowledge that is often known to the programmer. Some have suggested including (optional) declarations.[12]

APL2 as a parallel language

APL2 is an inherently parallel language because almost all primitive operations are defined on arrays of objects. The following sections classify and discuss these primitive operations. Akl defines parallelism as follows:

> Given a problem to be solved, it is broken into a number of sub-problems. All of these sub-problems are now solved simultaneously, each on a different processor. The results are then combined to produce an answer to the original problem.[20]

The key to exploiting parallelism is finding independent subproblems to be solved. The following discussion of each of the classes establishes how

Figure 1 Monadic scalar functions

Ceiling	Floor	Pi-times
Conjugate	Magnitude	Reciprocal
Direction	Natural logarithm	Roll*
Exponential	Negative	
Factorial	Not	

* See Reference 22.

Figure 2 Dyadic scalar functions

Add	Greater than or equal	Nand
And	Less than	Nor
Binomial	Less than or equal	Not equal
Circular functions	Logarithm	Or
Divide	Maximum	Power
Equal	Minimum	Residue
Greater than	Multiply	Subtract

Figure 3 Right scalar function

Index of

independent subproblems can be defined. This then gives the key to implementation of these operations on a broad spectrum of parallel machines. For example, these operations could be done one per processor on a SIMD machine, or assigned in groups (based on data location) on a MIMD machine.

Scalar functions. Scalar functions can be most easily defined as the ability of a function to operate on individual elements of an array in exactly the same way that they are applied to the entire array. In other words, the calculation of every individual data element is independent of the other.

The following paragraphs define in turn monadic and dyadic scalar functions. The discussion of dyadic scalar functions includes the concepts of scalar extension, and also introduces two new terms, right scalar function and left scalar function. The functions that fit each of these categories are listed. Finally, there are functions that are closely related to scalar functions but do not fit the strict definition. These are also presented.

Monadic scalar functions. The formal definition of a monadic scalar function[21] is any function that meets the following requirement:

```
(F R)[I] ↔ F R[I]
```

The heart of this definition is the fact that the calculation of each element is independent of any other and that the definition of the operation on the whole array is defined in terms of the operation of the function on the individual elements. The functions in Figure 1 are defined in APL2 as being monadic scalar functions.

Dyadic scalar functions. The definition of a dyadic scalar function[23] is very similar to the definition of the monadic case. A dyadic scalar function is any function that meets the following requirement:

```
(L F R)[I] ↔ L[I] F R[I]
```

Again the independence of the individual calculations can be seen. Figure 2 illustrates the dyadic scalar functions.

Scalar extension. Scalar extension in APL2 is defined as "If one argument is a scalar or a one-item vector, pair the scalar or one-item vector with each item."[24] This allows APL2 to express the concept implicitly that most parallel languages define explicitly as a "data broadcast." The advantage in APL2 is that the programmer does not need to express the broadcast as a separate operation.

Right scalar functions—Consider now the case that the left-hand argument is not a single item, so that scalar extension would take place, but rather a data structure that is needed by each application of the function to items in the right argument. Therefore what we desire is not a scalar broadcast, but rather an array broadcast. This concept is captured in the following definition. A function will be called a right scalar function if the following is true:

```
(L F R)[I] ↔ L F R[I]
```

Although the term and definition is new, the concept is already used in APL2 in the function shown in Figure 3.

Left scalar functions—In a similar way, any function that meets the following requirement will be called a left scalar function:

```
(L F R)[I] ↔ L[I] F R
```

Again, this concept is also already used in APL2 in the function shown in Figure 4.

Each. The operator **each** accepts a single function as an operand, and the resulting derived function is monadic or dyadic based on the valence of that function. **Each** changes the operation of the function such that the function, instead of being applied to the entire argument(s), is rather applied to each item of the argument(s). The combination of all of these applications is the result of the derived function. **Each**, when applied to any function, produces a derived function that is by definition a scalar function (see Figure 5).

However, to be applied in parallel, one additional criterion must be satisfied; each application of the function must be independent of the others. The practical implication of this is that the function that is used must be free of side effects. This is true of all primitive functions in APL2 except for **roll** and **deal**. But this is not true of user-defined functions in APL2 in general.

Scalar related functions. There are a number of functions in APL2 that, although not strictly scalar functions, still exhibit many of the characteristics of scalar functions. These are listed in Figure 6, and the following paragraphs provide a brief description of how they are related to scalar functions.

Find—**Find** can be defined in terms of the left scalar function **member**. Each item of the left argument is searched for in the right argument using the member. After each search the partial result is shifted to another processor, based on the shape of the left argument, and the next search done. Clearly this is a highly parallel operation.

Format—In all three **format** functions—default, **format by example**, and **format by specification**—there is a right scalar operation. In **format by example** and **format by specification** the formatting of each item in the right argument can be carried on completely independently of the other. Only when all of the items are formatted must the result be compiled to form a new matrix. However, even this operation is an operation that is performed along axes and can be done in parallel.

Figure 4 Left scalar function

Member

Figure 5 Scalar derived functions

Each (monadic)	**Each** (dyadic)

Figure 6 Approximately scalar functions

Bracket indexing	**Index**
Find	**Index with axis**
Format (default)	**Interval**
Format by example	**Without**
Format by specification	

In the case of default formatting, there is an additional step of determining the format parameters for each column. This also can be done in parallel with each processor determining on its own the required size for its item. These can then be combined together in a process very similar to reduction along the first dimension.

Indexing—Indexing, both in its functional form and as bracket indexing, would be difficult, but rewarding, to implement in a parallel form. **Index** would be considered a result scalar function, that is, each item in the result can be determined using an independent calculation based on the arguments. First, several sequential steps would be completed. The shape of the result would have to be determined and the locations allocated. Next, each location in parallel could obtain the correct indices; then, based on the shape of the array being indexed, it could determine positions and finally get the value from that position.

Interval—**Interval** is also a result scalar function. **Interval** would be very easy to implement on any machine in which each processor could determine a unique ID (identification), and all the IDs are sequential. **Interval** could then be simply imple-

Figure 7 Reduction functions

Reduce	**Reduce N-wise**
Reduce with axis	**Reduce N-wise with axis**

Figure 8 Scan functions

Scan	**Scan with axis**

Figure 9 Product functions

Decode	**Inner product**
Encode	**Outer product**

mented as laying out the shape of the result and telling each processor to generate a number based on its ID.

Without—**Without** is defined[25] as follows:

$$L{\sim}R \leftrightarrow (\sim L{\in}R)/L$$

The **member** and **not** part of the definition (most likely combined as a single operation) can be considered to be a left scalar function as defined above. The parallel nature of **replicate** will be dealt with later.

Reduction and scan. **Scan** and **reduce** operations (Figures 7 and 8), like scalar functions, have been at the heart of APL since its inception. Their importance to parallel processing has also been clearly established. Steele has called them primitive parallel operations.[5] **Reduce** can be considered a subset of the **scan** operation where only the final value is considered to be important.

When defined on vectors, these operations are parallel only when the function that is being applied is associative,[26] so only the associative case will be dealt with here. Brenner,[7] along with many others, has outlined a method of doing the **scan** (and therefore **reduce**) in $2{\circledast}\rho X$ passes. This means, for example, that a million element vector can be scanned in 20 operations on a sufficiently parallel machine.

The placement of these operations in this classification is very difficult. They are placed here so that we can deal with their primary definition, that is, on vectors. However, they are often used on higher order arrays (for example, on matrices). When applied to matrices these operations exhibit two levels of parallelism. First is the parallelism outlined above. Second is the parallelism that is involved in the application of the operation along an axis, as outlined in the next section. These two levels of parallelism can be handled separately, or combined to generate a highly parallel construct.

Product functions. The final set of functions that must be considered before we leave the area of scalar functions is the product functions (Figure 9). These functions are based on the dot operator.

Decode and **encode** are included with the product functions because they can be expressed in terms of the product functions.

The product functions are also result scalar functions, with each item in the result being calculated from a separate calculation. In the case of **outer product** this is the simple application of a function to two data items. In the case of **inner product** this result is more complex, consisting of the application of a function on two vectors and applying **reduce** to the resulting vector.

Axis functions. Moving from the area of scalar functions, the next logical step would be functions that are applied to subarrays of the arguments. These will be called axis functions. However, before presenting the individual functions, it would be helpful to formalize the concept of an operation along an axis and the concept of subarrays.

Subarray. A subarray is a subset of the data contained in an array that is selected by using zero or more elided axes. All nonelided axes must have a scalar value.

In APL2 the axis specification can be used to apply the function to independent subarrays within an array. The axis specified indicates the axis that is to be elided. We shall demonstrate this principle by discussing **enclose with axis** and **disclose with axis**. These functions were chosen because they can be used to describe all other operations that take an axis specification (see Table 1).

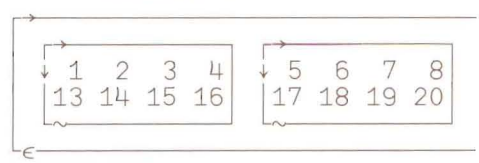
Table 1 Decomposition of axis specifications. The columns in this table show a decomposition of each of the lines of code that can be used to replace the most general case for each of the functions with axis specification.

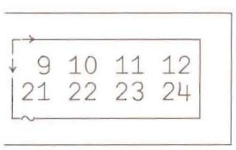
Function	Result	Disclose Operation	Enclose Operation for Left Argument	Operation	Enclose Operation for Right Argument
Catenate	Z←	⊃[A]	(⊂[A]L)	,⍤	⊂[A]R
Expand	Z←	⊃[A]		L\⍤	⊂[A]R
Partition	Z←	⊃[A]	(⊂L)	⊂⍤	⊂[A]R
Reduce	Z←	⊃		O/⍤	⊂[A]R
N-wise reduce	Z←	⊃[A]	L	O/⍤	⊂[A]R
Replicate	Z←	⊃[A]		O/⍤	⊂[A]R
Reverse	Z←	⊃[A]		⌽⍤	⊂[A]R
Rotate	Z←	⊃[A]	L	⌽⍤	⊂[A]R
Scan	Z←	⊃[A]		O\⍤	⊂[A]R
Drop	Z←	⊃[A]	(⊂L)	↓⍤	⊂[A]R
Index	Z←	⊃[A]*	(⊂L)	−⍤	⊂[A]R
Laminate	Z←	⊃[⌈A]	(⊂⍤L)	,⍤	⊂⍤R
Ravel	Z←	⊃[↑A]		,⍤	⊂[A]R
Scalar	Z←	⊃[A]	(⊂[A]L)	O⍤	⊂R
Scalar	Z←	⊃[A]	(⊂L)	O⍤	⊂[A]R
Take	Z←	⊃[A]	(⊂L)	↑⍤	⊂[A]R

* See Reference 32.

The **enclose with axis** function takes subarrays along the axes specified and makes them a single data item in the result. Therefore, the resulting matrix has the shape of the argument with the specified axes removed, and each item has the shape of the axes removed. For example:

```
      A←2 3 4⍴⍳24
      B←⊂[1 3]A
      ⍴B
3

      ⍴1⊃B
2 4

      DISPLAY B
```





The **disclose with axis** function is very similar to this except the elements are disclosed and placed back in the subarrays as specified. For example:

```
      C←⊃[1 2]B
      ⍴C
2 4 3
      C
 1 5 9
 2 6 10
 3 7 11
 4 8 12

13 17 21
14 18 22
15 19 23
16 20 24
```

For each of the axis functions listed in Figure 10, the application of the function results (conceptually) in the enclosing of the array along the given axis, applying the function to each item of the result, and then disclosing the result along the same axes. In light of parallel operation, it can be considered that each of the operations on the subarrays is an independent operation, and therefore can be done in parallel.

Scalar functions with axis. In addition to the above operations that take an axis specification, all of the scalar functions can take an axis specification. The concept is also based on subarrays and can be expressed in terms of **enclose** and **disclose** (see Table 1). The axis specification on scalar functions causes the items in one argument to be broadcast (scalar extension) to subarrays in the other argument.

Figure 10 Axis functions

Catenate	**Ravel with axis**
Catenate with axis	**Replicate**
Disclose	**Replicate with axis**
Disclose with axis	**Reverse**
Drop with axis	**Reverse along first axis**
Enclose with axis	**Reverse with axis**
Expand	**Rotate**
Expand with axis	**Rotate with axis**
Laminate	**Rotate along first axis**
Partition	**Take with axis**
Partition with axis	

Figure 11 Recursive functions

Depth	**Enlist**	**Match**

Figure 12 Matrix inversion functions

Matrix divide	**Matrix inverse**

Axis operators. Bernecky[27] and Gfeller[28] have both described a language enhancement called axis operator. Although their descriptions are different in syntax, they both carry the same fundamental idea. The axis operator has the effect of dividing the arguments into smaller matrices and applying the function to these smaller items. This type of operator would allow functions to be considered as axis functions, independent of their original type, much as the **each** operator forces its operand to be considered as a scalar function.

Recursive functions. Some functions in APL2 can be defined in the form of the following recursive definition:

f(x) = g(f applied to each item in x)

Where: f(x) is a function that is defined at some level in the tree (usually simple scalars)
g(x) is a combining function

The characteristic nature of these functions is that their execution results in a tree structure. The calculation in each of the branches of the tree is independent of the others and therefore can be done in parallel. The functions are shown in Figure 11.

Depth. The function **depth**, which returns the depth of the deepest item in a nested array, can be expressed in terms of a recursive definition:

$\equiv R \leftrightarrow 1+\lceil/\equiv{}^{\cdot\cdot}R$

Where: $\equiv$ *simple scalar* $\leftrightarrow 0$

Enlist. **Enlist** converts a nested array into a simple vector using a **depth** first method. The recursive definition of this routine is:

$\in R \leftrightarrow \supset,/\in{}^{\cdot\cdot}R$

Where: $\in$ *simple scalar* $\leftrightarrow$ *one item vector*

Match. **Match** returns a 1 if the two structures are identical at all levels, and a 0 otherwise. The recursive definition of this routine is:

$L\equiv R \leftrightarrow \wedge/L\equiv{}^{\cdot\cdot}R$

Where: $L\equiv R \leftrightarrow 0$ if L and R have different shapes
$L\equiv R \leftrightarrow 0$ if L and R are *simple scalars* with different values
$L-R \leftrightarrow 1$ if L and R are *simple scalars* with the same value

In general, the execution speed of **match** can be improved if, when any nonmatching condition is detected, all the execution in the tree is terminated and the 0 result returned. This makes the execution of the branches nonindependent, but they still can be executed in parallel.

Whole array functions. Moving from scalar to subarrays, the next logical step would be operations that manipulate entire arrays and therefore do not contain simple independent operations. However, both of these operations (see Figure 12) have been studied as classic parallel programming problems with many already published solutions.

The sorting functions in APL2 (see Figure 13) take an array as an argument and return a vector of indices as a result.

Rearrangement functions. The last class of operations that can be executed in parallel are those that

deal with data rearrangement. The characteristic of each of these functions is that the operation is done on addresses and not on data. The operations are shown in Figure 14.

The method of execution for each of these operations is basically the same:

1. Calculate the shape of the result.
2. Create an array of processors that match this shape.
3. Broadcast the control information to each processor.
4. Each processor calculates the current position of the data that are needed at that processor.
5. Each processor gets the data.

Not parallel. Some operations in APL2 cannot be executed in parallel. The primary reason for this is that they are defined on single objects or they do only a single operation. These operations are shown in Figure 15.

For example, **deal** is only defined on scalars. **Enclose**, **first**, **pick**, and **shape** all do a single operation on an entire array. **Execute** executes only one vector at a time. However, that line could be a parallel operation.

Other possible parallelism. There are other areas of possible parallelism in APL2. These are not discussed in this paper but are mentioned here for completeness.

Vector notation. Vector notation, or strand notation, allows a vector to be created by placement of objects next to each other. When these objects are simple constants, then creation of the vector is very straightforward. However, if the objects are expressions involving calculations, then this very simple construct allows for expression of a fork and join parallel structure.

Data-flow analysis. Data dependence is key to detecting parallelism in programs.[29] Several authors[12] have explored some of the areas of data-flow evaluation in APL. Most of this work has been related to the work being done on APL compilers.

Measurement of parallelism in APL2 code

For the measurements on the degree of parallelism, three applications were selected. These were selected to cover a broad spectrum of applications

Figure 13 Sorting functions

Grade down	
Grade down (w/collating sequence)	
Grade up	
Grade up (w/collating sequence)	

Figure 14 Rearrangement functions

Drop	**Transpose** (general)
Reshape	**Transpose** (reversed axes)
Take	

Figure 15 Functions that cannot be executed in parallel

Deal	**First**	**Ravel**
Enclose	**Pick**	**Shape**
Execute		

from the commercial data processing field. Each of the applications studied represents real code either available as a product or running in a manufacturing support area. Each of the applications is described briefly below, along with an explanation of the distinctions of that application.

Database application. The first application studied was a database verification process. In this process approximately 5000 database records are read and all of the data in those records are verified. The information is verified by checking for consistency against lookup tables and checking for conformance to established input formats. The database is also conditioned to conform to the requirements for later processing.

This application was selected to show APL2 working in a non-numeric processing intensive process. The processing involves a large number of searches, sorts, justifications, and merges.

Interactive application. An education catalog and enrollment system was selected as an example of an interactive application. This system was highly user interactive, being completely full screen and menu driven. Within the application all user input is

checked for errors. During the session studied, the users searched the catalog using two different methods, viewed two course descriptions, enrolled in a course, scheduled time in a learning center, and reviewed their current enrollments.

This program contains a large amount of control flow logic code, which decodes user commands and performs complex error checking. Also, since it is a full-screen design, it must create and refresh screens and windows. The application also does a significant amount of formatting of data to display in "nice" formats to the user. This application would be considered by most to be a highly sequential system.

Graphics application. The last of the three applications that was studied is the GRAPHPAK workspace that is distributed with APL2. This workspace does a variety of presentation, business, and scientific/engineering graphics. The DEMO program within this workspace was used for the measurements on this application. This code represents fairly old APL code (late 1970s) that was written long before any emphasis on parallel processing.

The GRAPHPAK workspace uses APL functions to manipulate vector represented images and display them using GDDM (graphical data display manager). It uses homogeneous coordinates to perform a number of scaling and rotation calculations on graphical images. It is a concentrated use of the numeric capabilities of APL2.

Description of method. To measure the data parallelism in APL2 it was necessary to collect statistics regarding the data passed as arguments and operands during actual APL2 operation. The method chosen for this was to replace every primitive function and operator call with a call to a function or operator that would produce the same results but would collect information regarding the data passed to the operation. This method is outlined below.[30]

Workspace conversion. The workspace conversion consisted of replacing each primitive function and operator call, and all uses of brackets with calls to user defined functions. Each of these replacement functions had to fulfill two distinct purposes. First, it had to do exactly the same data manipulation as the primitive function. Second, it had to collect data and save the data for future use (see the next section on data collection). The two actions must be totally isolated from each other.

The first part of the replacement operation is easy in most cases. Most of the time it is possible simply to call the function that is being replaced. However, there are some cases that present problems. The replacement functions must explicitly handle fill and identity functions for empty arguments. Also bracket indexing and bracket axis must be implemented using the syntax of normal functions and operators. Finally, the **outer product** must be implemented as a monadic operator.

The workspace conversion consisted of replacing each primitive function and operator call.

A set of conversion routines was created that replaced all the primitive operations, as listed above, to the replacement routines. Often this was a simple replacement, but sometimes it involved syntactic changes to the code. For example, all bracket indexing were converted to the **index** function.

The converted workspace was shown to be the functional equivalent of the original workspace through a variety of verification methods. This converted workspace could then be run and the data collected automatically during operation.

Data collection. Each replacement function also must collect data. Each function evaluates its arguments, summarizes the information based on the operation type, and passes the information to the $\Delta\,\Delta COLLECT$ function. The $\Delta\,\Delta COLLECT$ function is responsible for compiling the information using several global variables. It is important that the data collection function interfere as little as possible with the application workspace.

The data were collected using a tabular method. A three-dimensional array was created with each plane being the information for one of the primitive

Table 2 Database application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	4,371	3,763	353	16,384	3,904	14	4,096
AXIS-A	10	10	2	2	10	8	8
AXIS-V	6,005	3,121	105	16,384	2,222	32	2,048
DERSCAL	12	0			0		
DSCALAR	2,781	343	181	8,192	0		
MSCALAR	206	63	60	128	0		
NOTPAR	1,751	1,470	3	1,024	20	31	256
PRODUCT	1,137	1,133	37	8,192	1,061	1,310	524,288
REARRANGE	3,786	1,096	7,064	524,288	3,303	3,048	524,288
RECURSE	10	10	9	16	0		
REDUCE	174	157	61	2,048	65	82	2,048
SCAN	70	70	44	64	1	8	8
SORT	0	0			0		

Table 3 Interactive application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	10,370	5,156	239	16,384	4,770	27	256
AXIS-A	218	218	2	16	71	6	64
AXIS-V	30,428	13,027	17	4,096	2,125	45	1,024
DERSCAL	1,879	1,650	4	32	0		
DSCALAR	18,022	2,906	21	1,024	140	2	2
MSCALAR	3,355	1,214	10	32	0		
NOTPAR	17,360	4,944	55	16,384	3,611	25	4,096
PRODUCT	1,627	1,515	8	64	276	247	2,048
REARRANGE	15,465	14,246	331	16,384	11,493	406	16,384
RECURSE	3,122	2,690	248	16,384	371	2	3
REDUCE	1,475	1,221	17	64	161	183	512
SCAN	307	282	22	64	3	9	16
SORT	14	14	8	16	13	12	32

operations. The arguments to the function are tabulated according to their primary and secondary parallel dimensions as in the table. The data are then tabulated in the array in groups; 0–8 have their own group and after 8 they are grouped by powers of 2.

The data collection routine also collects data on routines that either do not fit the above method or require more information to be saved. These are called exception data. All of these data are gathered during the operation of the application and then saved when the program is done.

Data analysis. The data are summarized by groups of operations. For each group, the following data are calculated:

- Total calls to operation—The total number of times that the operations in the group were called during running the application

For both the primary and secondary parallel dimensions:

- Parallel operations—The number of times the given operation(s) were called with two or more data items
- Average data items—The average number of data items for all parallel calls
- Maximum data items—The maximum number of data items presented to this operation by any single execution

Table 4 Graphics application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	27,393	13,957	16	2,048	10,541	63	512
AXIS-A	651	651	2	2	637	52	1,024
AXIS-V	27,587	20,905	22	2,048	4,914	19	1,024
DERSCAL	0	0			0		
DSCALAR	80,584	15,416	16	2,048	0		
MSCALAR	3,663	1,275	47	2,048	0		
NOTPAR	9,484	4,764	2	1,024	5	28	128
PRODUCT	3,026	2,912	14	1,024	1,387	39	2,048
REARRANGE	21,620	13,195	47	2,048	13,142	40	2,048
RECURSE	0	0			0		
REDUCE	6,235	5,899	10	512	1,276	16	1,024
SCAN	205	172	9	128	135	6	16
SORT	284	0			284	48	512

Table 5 Overall application parallelism

Group Name	Total Calls to Operation	Primary Parallel Dimension			Secondary Parallel Dimension		
		Parallel Operations	Average Data Items	Maximum Data Items	Parallel Operations	Average Data Items	Maximum Data Items
ASCALAR	42,134	22,876	122	16,384	19,215	44	4,096
AXIS-A	879	879	2	16	718	47	1,024
AXIS-V	64,020	37,053	27	16,384	9,261	28	2,048
DERSCAL	1,891	1,650	4	32	0		
DSCALAR	101,387	18,665	20	8,192	140	2	2
MSCALAR	7,224	2,552	30	2,048	0		
NOTPAR	28,595	11,178	26	16,384	3,636	25	4,096
PRODUCT	5,790	5,560	17	8,192	2,724	555	524,288
REARRANGE	40,871	28,537	458	524,288	27,938	546	524,288
RECURSE	3,132	2,700	247	16,384	371	2	3
REDUCE	7,884	7,277	12	2,048	1,502	37	2,048
SCAN	582	524	21	128	139	6	16
SORT	298	14	8	16	297	47	512

Since the information for some operator calls was included in the exception data, this information is also summarized.

Results. The results for each of the applications above are summarized in the following tables: Table 2, database application; Table 3, interactive application; and Table 4, graphics application. Table 5 shows the combined results.[31]

General observations. The percentage of parallel operations (approximately 45 percent of the 300K+ operations) is high. The average number of data items is moderate, 10–100.

It is interesting to note that the array operations force the user to write array code, hence there is a very high percentage of parallel operations. However, the scalar operations, especially dyadic scalar functions, which allow the user to write scalar code, have a much lower parallel operations count.

Application-specific observations. The database application exhibits the highest percentage of parallel operations (56 percent) and the highest average parallel operations (as high as 7K). The graphics application has lower average parallel operations than might be expected. This might be due to the fact that when this system was written, machines were smaller, and looping solutions were often used where array solutions would be used today. The interactive solution exhibited a higher than expected degree of parallelism.

Conclusions

In the introduction to this paper, the question of how much parallelism is expressed implicitly in APL2 was presented. This study shows clearly that APL2 exhibits a high degree of parallelism in its structure. APL2 is a parallel language due to a historical perspective that placed a high emphasis on array operations. The paper establishes that 94 of the 101 primitive APL2 operations can be implemented in parallel. We demonstrate also that typically 40–50 percent of APL2 code in "real" applications is parallel code. In light of these statistics, it is clear that APL is already a powerful parallel language.

Language recommendations. There are some features of the language that reduce the available parallelism. The following recommendations address specific areas in the APL2 language that increase the potential for parallel operation.

Side-effect-free functions. It has been shown that **each** is a highly parallel construct that can be used as a fork-join construct. However, for defined functions this construct cannot be executed in parallel. This is the result of the lack of side-effect-free functions in APL2. It is necessary for the programmer in APL2 to be able to declare, or have the system detect, that a given function has no side effects. Once this is done, it will be possible to parallelize expressions involving **each** (and other operators) without worrying about data interference.

Axis specification on more operations. Only a subset of the APL2 operations currently accept an axis specification. Because of this it is necessary for the programmer to manipulate the data before and after the operation to make the data conform to the required axes. Often the programmer will use an explicit **enclose-disclose** pair, use **transpose**, or (worse still) write a looping solution. This problem could be solved in two different ways. APL2 could be modified so that all, or at least most, operations accept an axis specification. Or, as has been proposed by others, an axis operator could be introduced.

Control flow operators introduced. **Each** is the first of several necessary control flow operators, essentially implementing a FORALL construct. Other operations need to be introduced to perform other structured processing constructs. For example, looping, recursion, if-then-else, case, and the WHERE construct from FORTRAN 8X need to be included. In addition to the obvious data-flow simplification, the APL2 language with these constructs (given a proper implementation) would be a much easier language to read.

Parallelization of APL2. In addition to the language considerations, this study leads to some conclusions in the area of execution of current APL2 on parallel machines.

Emphasis on array functions. Much of the emphasis in parallelizing APL2 has been with the scalar functions. This study points out that the array functions also provide a rich resource for parallelization. By considering the rearrangement functions as parallel operations on the addresses of data, rather than the data themselves, a large pool of parallel operation is available.

Importance of data-flow analysis. Finally, it is important to note that although 45 percent of the calls in this study could be executed in parallel, there is still a large body of code that is sequential in nature. Data-flow analysis is the key to unlocking the parallelism in this code. Much emphasis must be placed on this area of research if APL2 is to prove a successful parallel programming language.

Cited references and notes

1. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962).
2. P. S. Abrams, *An APL Machine*, Ph.D. dissertation, Stanford University, Stanford, CA (February 1970), p. 64.
3. W. D. Hillis and G. L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM* **29**, No. 12, 1170–1183 (December 1986).
4. Abrams (see Reference 2) shows how to do some array manipulations without any movement or calculation involving the actual elements. This concept involved the use of what he called *array descriptors*.
5. G. L. Steele, Jr., "Design of Data Parallel Programming Languages," presented at Syracuse University, NY (December 7, 1988).
6. "American National Standard for Information Systems Programming Language FORTRAN," Draft S8, Version 112, June 1989, *FORTRAN Forum* **8**, No. 4 (December 1989).
7. N. Brenner, "APL on a Multiprocessor Architecture," *APL82 Conference Proceedings*, *APL Quote Quad* **13**, No. 1, 57–60, ACM, New York (September 1982).
8. W. D. Hillis, "The Connection Machine (Computer Architecture Based on Cellular Automata)," *Physica* **10D**, 213–228 (1984).
9. J. Delo, "A High-Performance Environment for APL,"

APL84 Conference Proceedings, *APL Quote Quad* **14**, No. 4, 122–129, ACM, New York (June 1984).
10. R. S. Clark and T. L. Wilson, "Vector System Performance of the IBM 3090," *IBM Systems Journal* **25**, No. 1, 63–82 (1986).
11. J. A. Brown, "An APL2 Description of the IBM 3090 Vector Facility," *APL88 Conference Proceedings*, *APL Quote Quad* **18**, No. 2, 44–48, ACM, New York (March 1988).
12. T. A. Budd, "Dataflow Analysis in APL," *APL85 Conference Proceedings*, *APL Quote Quad* **15**, No. 4, 22–28, ACM, New York (1985).
13. W.-M. Ching, "Automatic Parallelization of APL-Style Programs," *APL90 Conference Proceedings*, *APL Quote Quad* **20**, No. 4, 76–80, ACM, New York (July 1990).
14. J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* **21**, No. 8, 613–641 (August 1978).
15. A. Koster, "Compiling APL for Parallel Execution on an FFP Machine," *APL85 Conference Proceedings, APL Quote Quad* **15**, No. 4, 29–37, ACM, New York (1985).
16. H.-C. Tu and A. J. Perlis, "FAC: A Functional APL Language," *IEEE Software* **2**, 37–45 (January 1986).
17. M. N. Wakshull, "The Use of APL in a Concurrent Data Flow Environment," *APL82 Conference Proceedings*, *APL Quote Quad* **13**, No. 1, 367–372, ACM, New York (September 1982).
18. J.-J. Girardot, "The APL90 Project: New Dimensions in APL Interpreters Technology," *APL85 Conference Proceedings*, *APL Quote Quad* **15**, No. 4, 12–18, ACM, New York (1985).
19. J. A. Gerth, "Toward Shared Variable Events—Implications of *□SVE* in APL2," *APL83 Conference Proceedings*, *APL Quote Quad* **13**, No. 3, 265–274, ACM, New York (March 1983).
20. S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
21. *APL2 Programming: Language Reference*, SH20-9227-3, IBM Corporation (1988), p. 54; available through IBM branch offices.
22. **Roll** is defined as a scalar function and for most practical applications can be considered a scalar function. However, it is important to note that it does not strictly fit the definition. Consider the following example:

```
      □RL←5000
      ?(⍳100)[5 10 15 20 25]
1 7 1 6 21
      □RL←5000
      (?⍳100);5 10 15 20 25]
5 5 5 5 21
```

The problem with **roll** is that it is a function with side effects, so the order of calculation is important. In most cases, the above can be ignored because in each case five random numbers are generated, independent of the order of calculation. Although this effect can usually be ignored on sequential machines, the introduction of parallel calculation of random numbers is a much more complex problem. (See Reference 33.)
23. See Reference 21, p. 55.
24. *Ibid.*, p. 58.
25. *Ibid.*, p. 250.
26. This is not strictly true. For example:

```
      -/X ↔ +/X×(⍴X)⍴1 ¯1
```

So it can be seen that some functions (notably **subtract** and **divide**) can be rewritten into functions that are associative.
27. R. Bernecky, "An Introduction to Function Rank," *APL88 Conference Proceedings*, *APL Quote Quad* **18**, No. 2, 39–43, ACM, New York (March 1988).
28. M. Gfeller, "A Framework for Extensions to APL," *APL88 Conference Proceedings*, *APL Quote Quad* **18**, No. 2, 162–165, ACM, New York (March 1988).
29. M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Programming* **16**, No. 2, 137–178 (1987).
30. R. G. Willhoft, "A Tool for the Empirical Study of the Execution of APL2 Primitive Operations," *APL Implementer's Workshop*, Syracuse University, NY (September 11–14, 1990). (Includes a complete description of the method and code used for the study.)
31. R. G. Willhoft, "Parallel Expression in the APL2 Language," *APL Implementer's Workshop*, Syracuse University, NY (September 11–14, 1990). (Includes more complete test results.)
32. This disclose must allow simple scalars as elements, i.e., if the argument to disclose is simple, then do nothing.
33. O. E. Percus and M. H. Kalos, "Random Number Generators for MIMD Parallel Processors," *Journal of Parallel and Distributed Computing* **6**, 477–497 (1989).

Accepted for publication June 21, 1991.

Robert G. Willhoft *IBM Information Systems Division, 1701 North Street, Endicott, New York 13760.* Mr. Willhoft is currently an advisory engineer in Systems Analysis Engineering, IBM Endicott. He is currently working on his Ph.D. from Syracuse University with an expected graduation of December 1991. His dissertation topic is *A Parallel Language for the Expression and Execution of Generalized Parallel Algorithms*. Mr. Willhoft received his M.S.E.E. degree from Syracuse University in 1984 and his B.S. from Geneva College in 1978.

Reprint Order No. G321-5449.

The foundations of suitability of APL2 for music

by Stanley Jordan
Erik S. Friis

APL is commonly used in scientific and quantitative applications, such as engineering and finance, but there has been little acceptance so far in artistic and symbolic applications, such as music. This paper demonstrates the suitability of APL2, a dialect of APL, as a powerful tool for the building of music-oriented software. The interactive interpreter, flexible built-in primitive functions and operators, and the independence from the details of the hardware are attractive features for music programmers. With APL2, a user can interactively create and transform complex informational structures. Thus, it is not only a formidable language for implementing music software, but also a valuable notation for representing the music itself.

Today, most music software is written in traditional compiled languages, such as Pascal and C. Applications include Musical Instrument Digital Interface (MIDI) sequencers, patch editors, and librarians as well as computer-assisted composition, analysis, and education programs. Some may feel that the mathematical orientation of APL2 is not well suited for music, with music occupying a place outside of the world of numbers. This may be conditioned by previous experience in which images are mathematical. For example, in math class, a teacher probably illustrated an increasing continuous function by drawing a curve, rather than by singing an ascending glissando.

A growing awareness of the mathematical nature of music may force a rethinking of this perception. We have found the awesome mathematical power of APL2 to be one of its strongest suites for musical software. Much of musical structure is based on its quantitative features. Quantitative relationships between parameters of sound form the basis of patterns and groupings. Many of the parameters themselves can be ordered in perceptual scales. Berry[1] even goes so far as to contend that all of the significant parameters of music, including rhythm, texture, and tonality, work in conjunction to create variations in intensity—lines of growth, decline, and stasis over time. Berry claims that these variations in intensity are the primary determinants of musical form, and intensity is the quintessential quantitative parameter.

Like standard music notation, APL2 uses a character set that is iconic. Since musicians are accustomed to iconic notation systems, APL2 quickly becomes a comfortable working environment. In fact, the

iconic nature of the language has led some to refer to it as "the international road-signs of programming."

Suitability of APL2 for music

Smith[2] asserts that APL2 appeals to the right hemisphere of the human brain, which is specialized for holistic thinking. Users of APL2 are encouraged to think holistically, in part because operating on collections of data is, in general, no more difficult than operating on single entities.

Another feature that appeals to the right hemisphere of the brain is that one often visualizes the data structures and their transformations while programming in APL2. The flexible structure and syntax of APL2 conform well to the way most musicians conceptualize music. Smith also writes:

> . . . users of APL2 claim that it is the most powerful programming language in existence. Enthusiasts claim that with only a few lines of code, they can create what is unachievable in most other languages. Indeed, the impact of using APL is so substantial that active users often report [that] their entire thinking process has been transformed by use of the language.
>
> And yet critics claim the APL language is impossible to learn and hard to use. Can this be true?[2]

Lafore[3] addresses the question of the difficulty of learning a less-than-English-like programming language—in this case, C. Lafore's comments seem even more relevant to programming in APL2:

> When most people first look at a C program, they find it complicated like an algebraic equation, packed with obscure symbols. "Uh oh," they think, "I'll never be able to understand this!" However, much of this apparent complexity is an illusion. A program written in C is not much more complicated than one written in any other language, once you've gotten used to the syntax. Learning C, as is true with any language, is largely a matter of practice. The more you look at C programs, the simpler they appear, until at some point you wonder why you ever thought they looked complicated.[3]

With APL2 one can easily create and manipulate complex data structures. These data structures can be used for an enormous variety of representations of musical structures. APL2 comprises a powerful set of primitive functions and a concise syntax for using these primitives to transform data. Transformations are an important concept in music, in that they provide a way of relating one set of sounds to another or deriving one from another in meaningful ways. Perception itself utilizes transformations, and with a clear representation for music, many transformations that make sense mathematically or structurally also make sense musically.

Most programming languages allow for the access and manipulation of a single piece of data at a time, such as a character, an integer, or a floating-point number. This observation is further apparent in the following text from Lafore:

> This is a rather amazing capability when you think about it: when you assign one structure to another [structure, in this case, refers to a C structure as opposed to a data structure in general], all the values in the structure are actually being assigned, all at once, to the corresponding structure elements. Simple assignment statements cannot be used this way for arrays, which must be moved element by element.[3]

Unlike C and most other programming languages, in APL2, operation on an entire structure is the rule rather than the exception.

Parallelism. There has been much discussion regarding parallel hardware in the computing literature. Many see it as the wave of the future—just a matter of time. This bodes well for music programmers, because music is highly parallel. The question is: What languages can be run on a parallel machine?

Most languages in use today were written for a machine using the Von Neumann architecture,[4] i.e., a single central processing unit capable of executing only one instruction at a time. Complex problems must be analyzed into their constituent parts in order to be solved. Obviously this can be a necessary and even essential component to problem solving. However, analysis is only helpful to a certain point. Beyond that, one could further granularize the problem, but further analysis will not result in the understanding or solution of the problem. Users of many programming languages are required to analyze a problem far beyond the level that clear human comprehension requires. For the sake of the computer, excruciating details of the computation must be specified. Users of most languages do not realize how much the computer is programming

them. Despite great advances in hardware capabilities, this situation has not changed much because most are still dealing with the limitations of a Von Neumann machine. To a remarkable extent the Von Neumann organization of our machinery still influences high-level language design.[5]

APL was designed without the typical constraints of the Von Neumann mind-set. It was first designed as a short-hand notation for describing algorithms and was only later implemented as a computer language. With conventional programming languages, programmers are constantly dealing sequentially with collections of data or operations on them that they actually think of as simultaneous. The ability when using APL2 to extend the domain of a program from individual elements to collections of elements without an increase in syntactic complexity, allows a more accurate representation of the holism that is being conceptualized. And not only do we naturally tend to group collections into gestalts, or wholes, but also we often change our scheme of organization at a moment's notice. APL2 also excels in this area.

Unconstrained environment. Whereas most programming languages force the making of many initial decisions regarding the data and program, APL2 lets you improvise. Since APL2 is interpreted, you can enter an expression and it will be executed immediately. Without those tiresome edit, compile, and link cycles, you are free to experiment with ideas and variations on ideas. And because data are in the active workspace and are always accessible, you can inspect the results of a single expression to make sure that it does what you intended. Satisfied, you can move on with confidence to the next step.

In APL2 there is no need to declare variables, define pointers, or allocate storage. You are free to change a variable at any time to any size, structure, or content without concern regarding where and how it will occupy memory. The APL2 interpreter will make these determinations by using a dynamic memory allocation scheme.

The late binding of APL2 expressions allows references to be made to names that do not as yet exist when a function or operator is defined, as long as the name exists at execution time. The workspace concept allows for the blending of applications at an atomic level, achieving an extensive level of integration. The result of all these features is an "ideal" environment consisting of arrays and a powerful arsenal of tools to manipulate them. This can be quite useful to musicians who are interested in addressing a particular problem, but who may not have the patience or interest in performing optimizations of the solution.

Notational simplicity. APL2 is extremely concise. If abused, this feature can lead to incomprehensible programs—if used properly, it can lead to a degree of clarity of understanding that puts APL2 in a class by itself. Such an advantage is familiar to mathematicians, who tend to simplify notation in order to clearly express complex ideas. In carrying less "notational baggage," one can concentrate more clearly on the concepts being represented or the relationships between them.

More verbose notations, such as those using keywords to represent built-in functions, are appropriate for concepts that are less often used. Keywords are helpful because of their associations to common words or concepts making them easier to remember. But for frequently-used concepts, people have a tendency to abbreviate—to choose shorter symbols. This is especially apparent in representing music. The fundamental concepts of APL2 are so repeatedly useful that they merit symbolic representation. We feel that history has in fact confirmed this. Despite ongoing language development and conflicts over standards, the core of the APL language has remained remarkably stable. The ultimate proof of the clear organization of the language is the ease with which it has been generalized.[6]

We believe that the symbols of APL2 were carefully chosen for their mnemonic value, making them surprisingly easy to remember. The conciseness of the notation seems to make it possible to view an expression and simultaneously see the "forest" and the "trees." User-defined terms in programs, which by nature are more variable, are represented by keywords, while the stable APL2 primitives remain symbolic. APL2 symbols also provide the additional benefit of avoiding name conflicts with user-defined terms. However, if one insists on using keywords, simple user-defined "cover functions" may be defined that call the primitives. This raises an important distinction that novices are not always aware of, namely, APL2's primitive functions and operators are independent of the symbols that represent them. Typically, a single symbol can call either of two functions depending on syntax.

Applicability to music

Having discussed the character of the language, we now discuss some examples of APL2 in the context of music software. The examples are simple, and they do not pretend to represent all the parameters of real music, but they are meant to serve as a guide to what is possible.

Pitch. Pitch is the psychological correlate of frequency, which most people conceptualize as a one-dimensional quality; however, our perception of pitch is actually two-dimensional. Research has shown that there are two psychological attributes of pitch, *tone height* and *tone chroma*.[7]

Tone height is simply the sensation of "highness" or "lowness." Tone chroma is the perception of note color regardless of octave. Babbitt coined the term *pitch-class* to refer to sets of octave-related pitches, where class refers to our sensation of equivalence of pitches so related.[8]

Tone height is particularly important in the perception of melodic *contour*—the shape of a melody as its ascending and descending patterns unfold. Tone chroma is especially important in harmony. When the *voicing* of a chord is changed by disposing its notes into new octave ranges, there is often a sense that its character has changed more texturally than harmonically.

To represent pitch in APL2, two values may be used—octave and pitch-class—so as to have separate control of these two psychological variables. On the other hand, the use of a single value often makes calculation easier. Thus it can be advantageous to use a single value for an internal representation, and two values for an external representation to the user for display and entry purposes.

The following examples describe a few methods for representing pitch in APL2:

1. One value—a frequency number expressed in cycles per second

```
FREQUENCY←220 155.57 92.5
```

2. One value—a MIDI note number

```
PITCH←57 51 42
```

3. Two values (pitch-class and octave) as a character vector

```
PITCH←'A3' 'Eb3' 'F#2'
```

4. Two values (pitch-class and octave) as a mixed vector

```
PITCH←('A' 3)('Eb' 3)('F#' 2)
```

5. Two values (pitch-class and octave) as a numeric vector

```
PITCH←(9 3)(3 3)(6 2)
```

Example 5 indicates a common method for indicating pitch-class, using an integer in the range 0–11 as follows:

Integer	Pitch-Class
0	C
1	C♯ or D♭
2	D
3	D♯ or E♭
4	E
5	F
6	F♯ or G♭
7	G
8	G♯ or A♭
9	A
10	A♯ or B♭
11	B

This system, first introduced by Babbitt,[8] uses modulo-12 arithmetic to reflect the cyclic nature of pitch-class relations. By using the MIDI conventions, one can express pitch in a convenient method. MIDI is a communications protocol that electronic musical instruments such as synthesizers and computers can use to send and receive real-time performance information. A MIDI *note number*[9] is a single integer (0–127) representing a key on a MIDI keyboard. This system assigns to middle-C the value "60." The "C#," a semitone higher, corresponds to the value "61." Thus, MIDI represents pitches indirectly—not as soundwave frequencies, but as key numbers on a very long keyboard (about ten octaves).

MIDI note numbers are convenient because they simplify calculation. For example, to transpose an array of pitches up a perfect fifth (seven semitones) one could simply enter:

```
PITCH←PITCH+7
```

Monophonic scores. A musical score is a notated representation of music, or a precise set of instruc-

tions to a performer. In this latter view, the comparisons to a computer program should be obvious. The musical vocabulary of today is so vast, and so varied, that composers often find that the traditional "common practice notation"[10] cannot always express what they have in mind. They have been forced to invent new notation systems that may take the form of written instructions to the performers or some new visual representation. In the field of electronic music, composers have an unprecedented degree of precision in the control of parameters of sound, such as *timbre* or tone color. How does one notate this? The answers may vary considerably, but many composers agree that the whole idea of notation, or more generally representation, has become a field of study in itself. The computer has been recognized as having the potential to bridge this gap, for it has the power, as Papert has pointed out, "to concretize the formal."[11] Anyone who has used a modern MIDI sequencer with a graphical interface can attest to this fact. But while most MIDI software on the market provides fixed representations that have proved useful and easy to learn for most people, there remain some who would like the power to create new representations, without committing to any one until it has proven its usefulness. Furthermore, although most good sequencers allow global editing and some degree of algorithmic generation, this generally takes the form of supplying parameters to fixed routines. Serious computer musicians require a programming language that extends easily from individual notes to higher-level descriptions. This is appropriate because composers typically think in high-level terms—often the exact notes are just the details. If a composer can specify structures at an appropriately high level, then the system becomes a much more useful tool of thought. We have found APL2 to be an excellent language for prototyping representations for music. Arrays in APL2 may be viewed as visual structures that can be formed and transformed with ease.

The discussion that follows illustrates some simple score representations and a few techniques for manipulating them. Many functions that apply to single pitches also apply to structured collections of pitches with little or no change in the syntax.

A monophonic score or melody can be represented by a simple numeric vector:

```
SCORE←60 62 64 65 67 69 71 72
```

Since this collection can be conceptualized as an individual entry, it can be described using APL2's vector notation and assigned a name in one step. The variable *SCORE* can represent either a semantic or syntactic musical structure. If the structure is regarded outside of time, then the vector—a semantic structure—may represent a collection of ordered pitches. The pitches do not have to be played at any particular starting time or tempo. On the other hand, if the vector is a syntactic structure, then it represents a series of ordered pitches with a temporal attribute. More detail can be found in Reference 12.

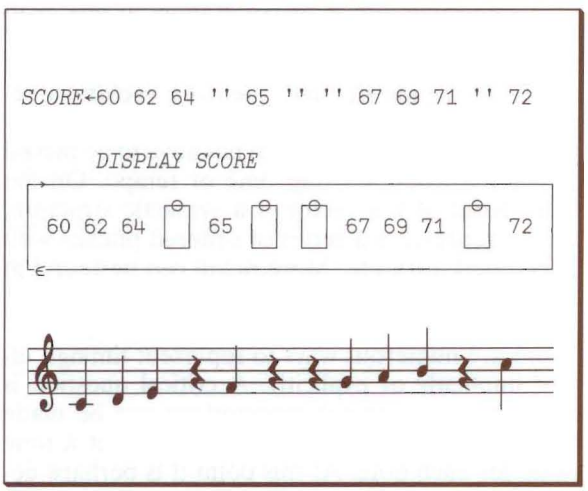
There are numerous ways to represent timings, either implicitly or explicitly. A critical question is whether some general assumptions can be made regarding timing, or whether to represent a time value for each note. At this point it is perhaps desirable to implement the latter approach since the timing information is varied and unpredictable, whereas the former approach is preferred when timings are more likely to be regular and predictable. At any rate, any decisions about definite timings are postponed until later, so we notate only an ordered collection of pitches.

An implied tempo can be defined such that each position in the array represents a beat, such as a quarter note or eighth note. Thus, there is a mapping between the index position in the vector and the order position in the pitch succession. If desired, an APL2 variable can hold a value for the duration of time represented by one step in the index position. Even varying tempos may be defined—i.e., the first four positions of the array represent quarter notes, or the next four positions represent eighth notes. But for this simple illustration, let us assume a constant time per index position.

Rests can be indicated by pairs of single quotes (with no spaces in between), to be used as "place holders" indicating empty elements. Figure 1 is an example of a score with three rests, the second one occupying two time periods.

Thus, if a constant time period is assumed, it is fairly easy to verify the timing simply by visually inspecting the score array. All manner of rhythms can be created in this way. If each index position corresponds to a much shorter duration, e.g., 64th notes, this representation has a much higher resolution, i.e., there is finer control of timing, but the rhythms will become less intuitively obvious. The

Figure 1 A simple representation of a score



result is a score that will occupy more space, both on the printed page and in computer memory. Such a representation will quickly become wasteful to the extent that the music is sparse, i.e., there is a high ratio of rests to notes. In this case, it is more economical to specify an explicit time value for each note, so as to make it unnecessary to account for the time periods between notes.

Suppose we wish to represent the melodic line shown in Figure 2. Pitches and start times can be assigned in two separate steps as shown in the figure, and can be put together into a single structure as follows:

```
SCORE←⍉⊃PITCHES TIMES
DISPLAY SCORE
```

```
↓60  0
 64  1
 69  2
 71  3
 72  6
 67  8
 65 16
 62 17
 64 18
 65 19
 74 22
 72 28
 71 29
 69 30
 71 31
 72 32
```

The variable *SCORE* contains a matrix where each row represents a single note. The first column of the matrix represents pitch and the second column represents starting time. We decide on eighth notes as the unit for timing. Thus in 4/4 time, there would be eight eighth-note time intervals per measure.

Now if we wish to add another parameter, loudness, we can define seven levels as variables using vector assignment

```
(ppp pp p mf f ff fff)←⍳7
```

where ⍳7 is shorthand for the series

```
0 1 2 3 4 5 6
```

and where □*IO* has been set to zero.

Figure 2 An example of pitch and time points

```
PITCHES←60+0 4 9 11 12 7 5 2 4 5 14 12 11 9 11 12
TIMES←0 1 2 3 6 8 16 17 18 19 22 28 29 30 31 32
```

These numbers are arbitrary. They are not intended as actual measurements of loudness intensity, but are used here to distinguish among seven graduated levels.

Initially, a third column is appended to the matrix, and each element in this column is set to the value "4." Now using the variable *f* assigned to value 4:

```
      SCORE←SCORE,f
      DISPLAY SCORE
```

```
↓60  0 4
 64  1 4
 69  2 4
 71  3 4
 72  6 4
 67  8 4
 65 16 4
 62 17 4
 64 18 4
 65 19 4
 74 22 4
 72 28 4
 71 29 4
 69 30 4
 71 31 4
 72 32 4
```

If we now want to retain the current timings, but reverse the order of the pitches from last to first—called a *retrograde*—this is very easy to specify in APL2:

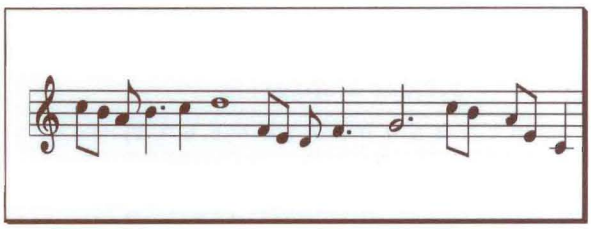
```
SCORE[;0]←⌽SCORE[;0]
```

The columns (and the rows) of the matrix are indexed starting from zero. Figure 3 shows the actual staff notation for this example:

```
      DISPLAY SCORE
```

```
↓72  0 4
 71  1 4
 69  2 4
 71  3 4
 72  6 4
 74  8 4
 65 16 4
 64 17 4
 62 18 4
 65 19 4
 67 22 4
 72 28 4
 71 29 4
 69 30 4
 64 31 4
 60 32 4
```

Figure 3 Reversing the order of the pitches



Now, suppose we want to edit the score in order to accomplish a specialized task—to find each note that occurs on a downbeat and make it one unit louder. The next section discusses the evaluation of the APL2 expression that accomplishes this task.

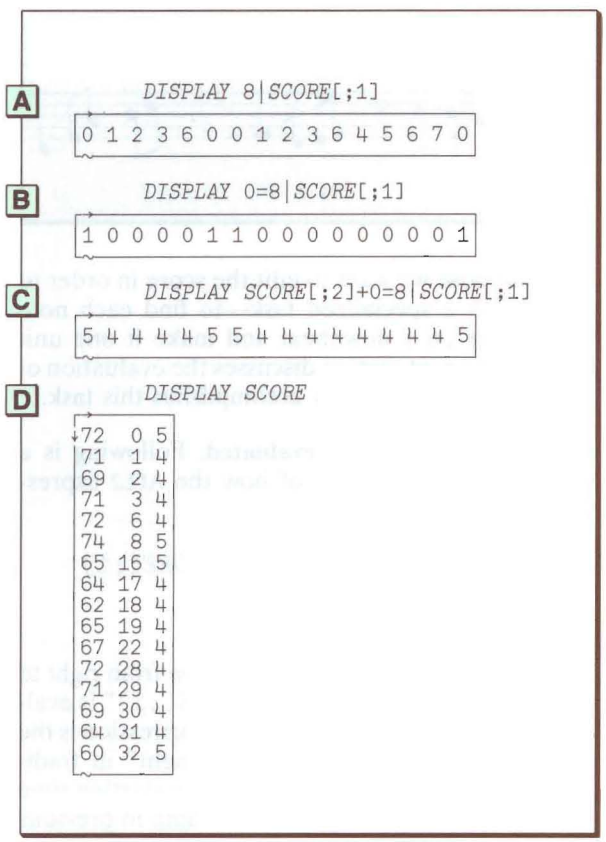
How the expression is evaluated. Following is a step-by-step explanation of how the APL2 expression

```
SCORE[;2]←SCORE[;2]+0=8|SCORE[;1]
```

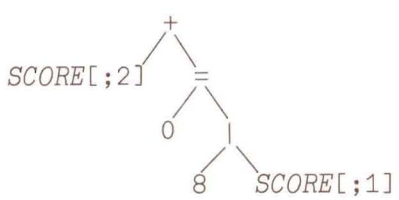
is evaluated:

1. Since APL2 evaluates an expression from right to left, the subexpression "8|*SCORE*[;1]" is evaluated first. The result of this subexpression is the eight-residue (or the "modulo-eight" in traditional mathematical terms) of the starting time values, shown in the second column in previous displays of the *SCORE* matrix. The argument "eight" for the **residue** function was chosen because each measure is eight time units long. See Figure 4A.
2. The next subexpression to be evaluated is $0=\omega$ where ω represents the result of 8|*SCORE*[;1]. The result of this subexpression is a Boolean vector whose corresponding elements are set where zeros occur in ω, which happen to fall on the downbeats. The "=" in APL2 is not an assignment, it is a test, returning a "1" when elements are equal in value, and a "0" otherwise. Figure 4B shows the resulting Boolean vector.
3. Next, the vector in Figure 4B is added to the loudness values (shown in the third column of previous displays) resulting in another intermediate value. Boolean values are numeric values and can be treated as such, illustrating a common use in APL2 programming, as well as the conciseness of the language. Figure 4C displays the result.
4. Finally, the result replaces the contents of the loudness values of the *SCORE* matrix and can be

Figure 4 Evaluation of APL2 expressions



seen in Figure 4D. A graphic representation of the parse tree for the same APL2 expression follows:



Our task description in natural language is translated into a one-line APL2 expression. Most other computer languages would have required many more lines of code and may have involved writing a program, compiling it, and linking it. This particular APL2 expression is not difficult to understand. In most other languages the solution would have been more complicated, simply because the extra lines of code and the loops would not contribute to the conceptualization of the process.

Polyphonic scores. So far we have only represented monophonic scores—that is, one voice, or "one note at a time." A polyphonic score represents more than one voice playing simultaneously. To represent a polyphonic score the monophonic model can be expanded by the introduction of rank or depth. In the previous section we started with a monophonic score represented as a numeric vector. A vector in APL2 has a rank of one, whereas a rank-two array is called a *matrix*. A matrix can be used to model a polyphonic score, such that each row or column of the matrix is the equivalent of a monophonic score. Since a matrix in APL2 must be rectangular and its rows and columns are parallel along each axis, the same ordinal and temporal attributes that formed the basis of the monophonic vector model also hold true for the polyphonic matrix model.

Figure 5 contains an example of a 3 × 8 matrix, which represents a polyphonic score. If we assume that each column in the matrix represents a quarter-note beat, this score represents eight major triads at quarter-note intervals, as expressed by the *SCORE* expression.

Notes in the same column are to be played simultaneously, and notes in the same row are to be played sequentially. Thus, we define a mapping between matrix dimensions and musical dimensions, such that each column is a time period, and each row is a voice. Of course the roles of the dimensions could be reversed. It is just a question of how we wish to visualize the structure. Changing the actual matrix to reflect this new mapping of parameters is simply a matter of applying the APL2 **transpose** function (not the musical transpose) to the array.

```
SCORE←⍉SCORE
DISPLAY ⍉SCORE
```

```
60 64 67
62 66 69
64 68 71
65 69 72
67 71 74
69 73 76
71 75 78
72 76 79
```

It is possible to imagine many mappings of this kind, all having different characteristics, and all being useful for different purposes. This reveals one of the reasons why the computer is such a powerful

tool for music: It provides the ability to create new kinds of musical representations as well as the freedom to explore the representations themselves as fields of interest.

A PLAY function. The potentials of computer music go beyond just representation. When linked to appropriate audio signal generating hardware, the computer can become a musical instrument with exciting capabilities. The discussion that follows illustrates the capabilities of a PLAY function that could be created in APL2. Define PLAY such that:

1. It accepts a score as a right argument.
2. The duration of each note will, unless otherwise specified, default to a set value, e.g., a quarter note.
3. An optional left argument may be accepted that specifies a common duration for all the notes, or a list of durations—one for each note.

The elements of the score are MIDI note numbers. The PLAY function sends performance instructions to an attached external device, such as a musical synthesizer.

As was previously illustrated, the following vector can represent a C-major scale:

```
SCALE←60+0 2 4 5 7 9 11 12

      DISPLAY SCALE
┌→──────────────────┐
│ 60 62 64 65 67 69 71 72 │
└∈──────────────────┘
```

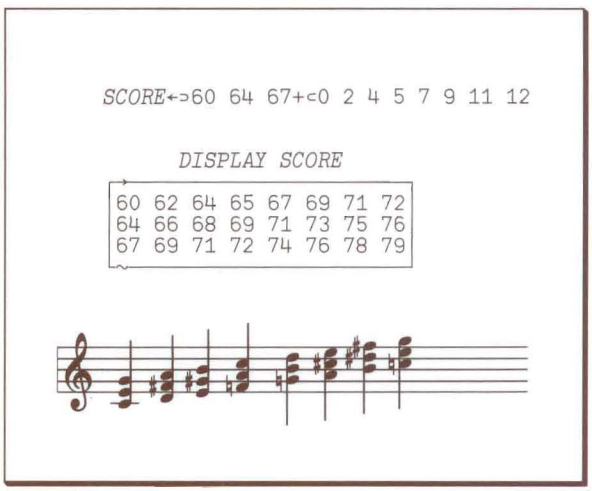
This expression will play the C-major scale starting at middle-C in ascending order:

```
PLAY SCALE
```

Since APL2 notation can be easily adapted for parallel processing models, it is interesting to examine the musical possibilities of a truly parallel version of the language.

For example, the **each** operator (¨) provides a formidable vehicle for exploring the parallel potentials of APL2. The **each** operator applies a specified function to each element of its arguments. Assuming a truly parallel **each** operator, when it is executed, envision a set of n independent processes running on a parallel multiprocessor, where n is the number of elements at the first level of depth in the array arguments.

Figure 5 A polyphonic score



A chord, which is a simultaneity of pitches, could then be played as follows:

```
MAJOR←0 4 7

PLAY¨60+MAJOR
```

Again, imagine three independent processes, each of which plays a single pitch for a set duration:

```
PROCESS 1     PROCESS 2     PROCESS 3
┌─────────┐   ┌─────────┐   ┌─────────┐
│ PLAY 60 │   │ PLAY 64 │   │ PLAY 67 │
└─────────┘   └─────────┘   └─────────┘
```

Depth can play an important role in the modeling of a musical score. For example, an increase in depth can signify that grouped notes are to be performed simultaneously.

```
60 62 64 (65 69 72) 67 69 71 72
```

The above vector is a polyphonic score representing three individual notes, followed by a chord inside of the parentheses, followed by four individual notes, all at a quarter-note tempo.

A further increase in depth could signify a set of virtual tracks to be played simultaneously, or sets of MIDI events on different MIDI channels. An increase in depth again could be used to model a set of multitrack tape decks or a set of MIDI cables.

Figure 6 An example of 12 major chords

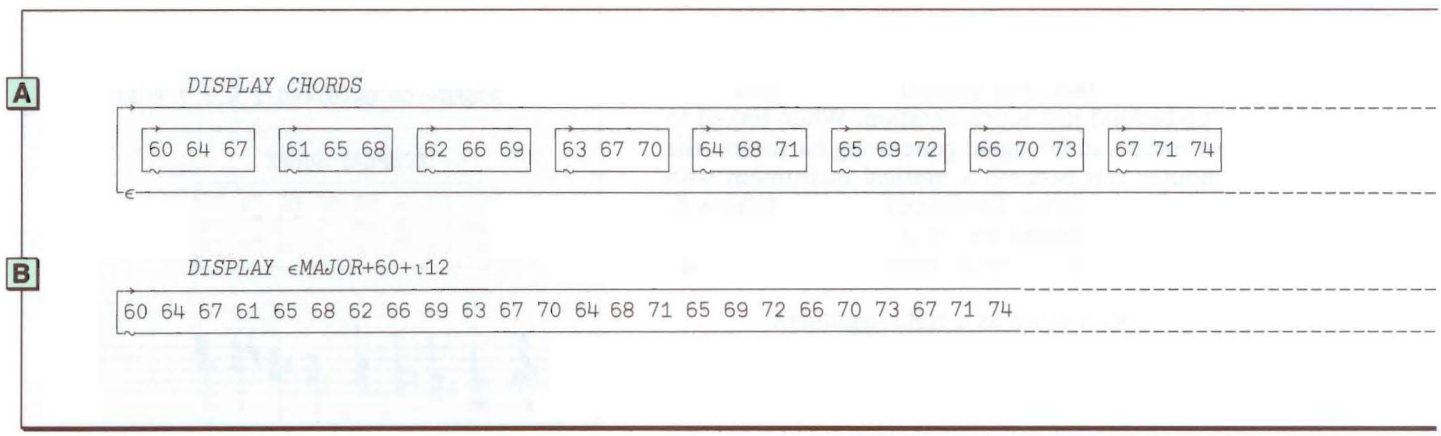
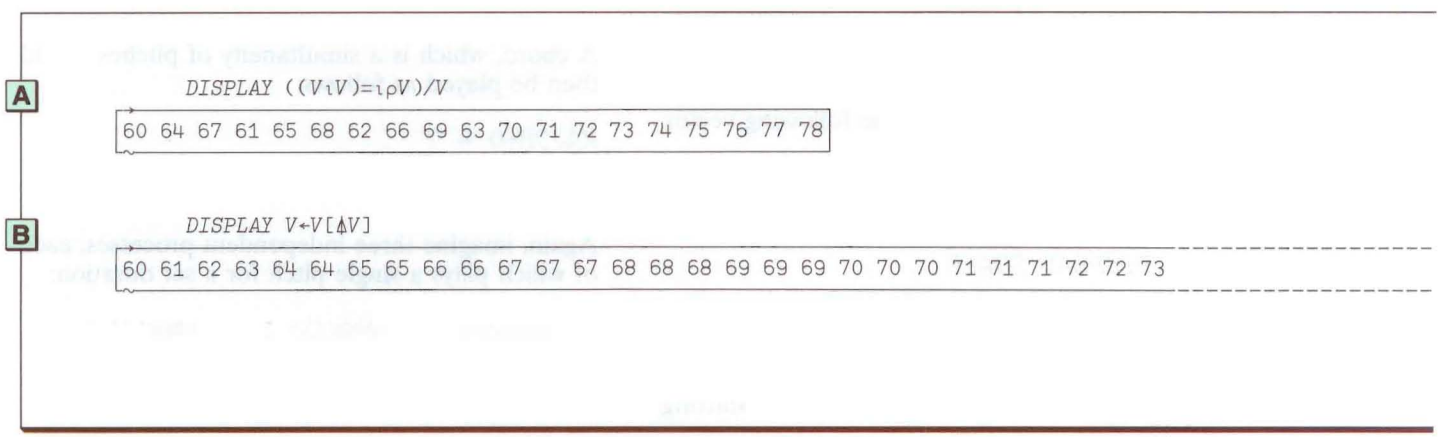


Figure 7 MIDI pitches with duplicates removed and sequential ordering



Utilizing depth, a sequence of 12 chromatically ascending major chords can be represented by:

```
CHORDS←60+(⊂MAJOR)+⍳12
```

and expanded as shown in Figure 6A. The following expression will play a sequence of 12 major chords:

```
PLAY CHORDS
```

while application of the **each** operator yields a sequence of three 12-note chords (each chord will sound quite complex):

```
PLAY¨CHORDS
```

The introduction of a second **each** will result in the playing of a chord constructed of the pitches represented by the MIDI note numbers 60 through 79:

```
PLAY¨¨CHORDS
```

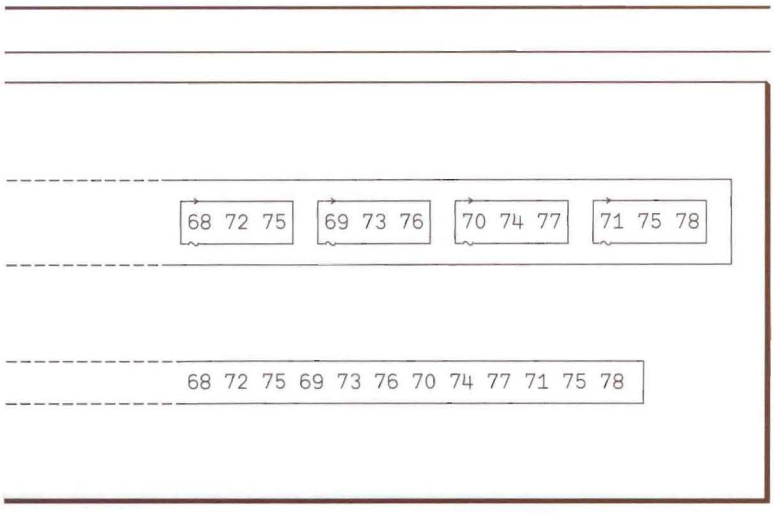
which is equivalent to

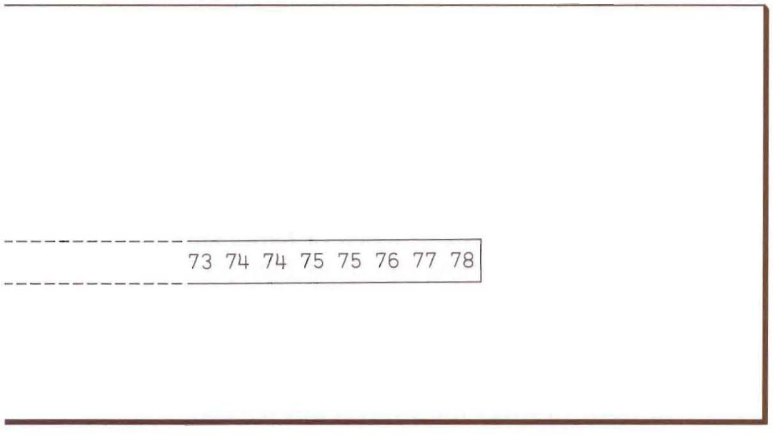
```
PLAY¨∈CHORDS
```

and

```
PLAY⊂∈CHORDS
```

Whereas the following expression will result in the





arpeggiation of the 12 chords:

```
PLAY∈CHORDS
```

Figure 6B displays the values of the generated pitches.

Note that some of the pitches represented in the resulting vector in Figure 6B have multiple occurrences, i.e., the same pitches occur in different arpeggios.

One extension that can be made to this model is to specify that a pitch can be played at different volume levels, determined by the number of its occurrences. A pitch that has no occurrences in the vector will not be played, or can be thought of as being played at a volume level of zero.

For example:

```
PLAY⍤⍤60 60
```

or

```
PLAY ⊂60 60
```

will sound one unit louder or perhaps twice the intensity of:

```
PLAY 60
```

If only unique pitches are to be selected from Figure 6B, then application of the following APL2 idiom to the vector of MIDI note numbers will filter any duplicates:

```
((⍵⍳⍵)=⍳⍴⍵)/⍵
```

Therefore, when this idiom is applied to:

```
V←∈CHORDS
```

the unique pitches are evaluated and shown in Figure 7A.

Or the pitches can be sorted by MIDI note number and can then be played sequentially at their relative volumes:

```
V←V[⍋V]
```

Figure 7B shows the ordered note numbers and can be played with the following:

```
PLAY VV←V⊂V
```

resulting in the groups shown in Figure 8A.

The relative volume of each pitch can be obtained by using:

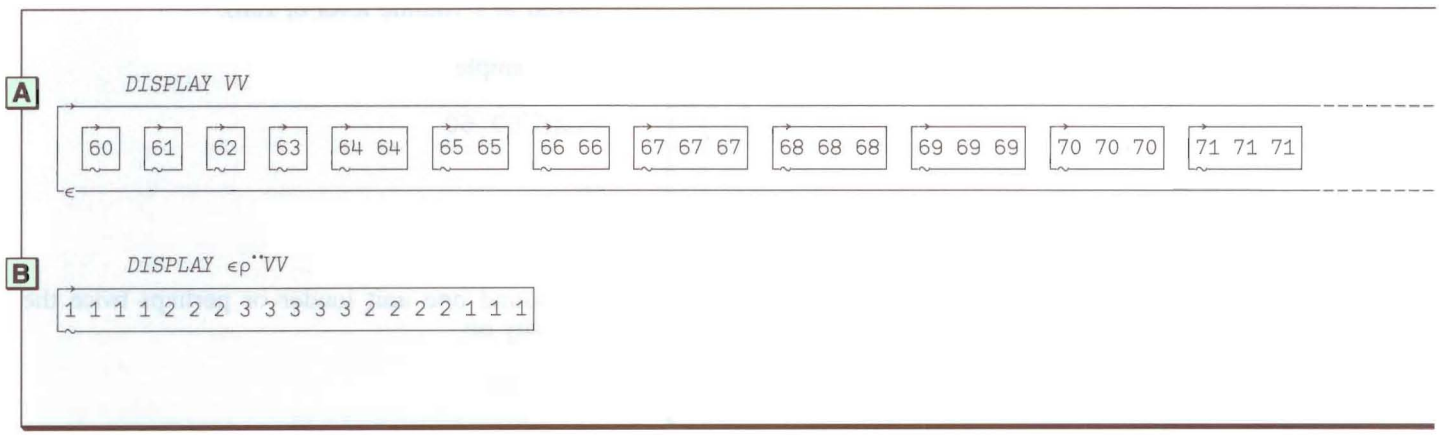
```
∈⍴⍤⍤VV
```

displayed in Figure 8B.

Finally, the following vector represents the number of distinct pitch-classes present in V:

```
⍴VV
```

Figure 8 Pitch and volume for ordered groups of note numbers



where

```
      DISPLAY ρVV
┌──┐
│19│
└──┘
```

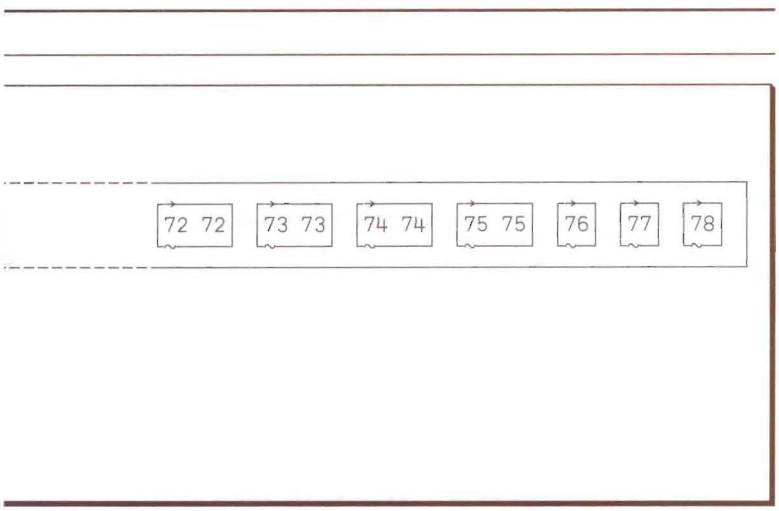
As can be seen, the power of APL2 to represent the music score in terms of pitch and volume is only the beginning of the use of computers in music applications.

The Smoliar model. We now describe another musical application of APL2. This work is inspired by Stephen Smoliar, who described a system for automated musical analysis.[13] Smoliar was himself inspired by Heinrich Schenker (1868–1935), who is widely regarded as the most influential music theorist of the 20th century.[14] Smoliar has designed a computational model loosely based on Schenker's theory of tonality. To enhance the understanding of the application, we first present some background in Schenkerian theory.

Heinrich Schenker's influence on music essentially corresponded to Noam Chomsky's transformational grammar[15] in the field of linguistics, although Schenker's work predated Chomsky's by a number of years. The similarities are striking. In both Schenkerian analysis and transformational grammar, a stream of symbols is scanned and recursively parsed into groups, yielding a hierarchical structure. Thus, a tree representation of a composition can be created where each level summarizes the events in the level below, from a higher-level perspective. The theory includes a suite of transformations, or *rewrite rules*, that can be used to alter the material without essentially changing its "meaning." Syntax is modeled by *trees*, and it is the rewrite rules that assert relations between trees, the most notable relation being similarity of meaning.

Smoliar writes, "Schenker viewed every well-composed tonal piece as being reducible to one of essentially three patterns, all based on the tonic scale and triad."[13] Before Schenker, much of harmonic analysis consisted of labeling chords as they progressed. This can lead to a concise harmonic description of the surface structure of a piece of music, but it does not adequately deal with the range of tonal functions each chord actually serves in context, or the range of structural levels at which it may function. Schenker asserted that the same kinds of voice-leading relations that exist from note to note, or from phrase to phrase, also hold true in the large-scale form of a composition, where entire sections or movements combine into a unified whole. A theory that is independent of structural level leads to a very elegant and organic view of musical structure.

Smoliar showed that many rewrite rules can be precisely formalized, so that a formal programming language of transformations can be developed. One can imagine computer-assisted analysis and computer-assisted composition programs that could provide new insights into the nature of tonal structure, perception, and the creative process itself. Smoliar's model was designed to assist music the-



orists in tonal analysis by representing music in a hierarchical structure and by effecting transformations that explicitly deal with this structure. A musical event is modeled as a tree structure, which can be entered or displayed at a computer terminal or internally stored as a list.

There are three types of events:

- A single note
- A sequence (SEQ) of events occurring in a designated order
- A simultaneity (SIM) of events

The structure is recursive because an element of an event can itself be an event.

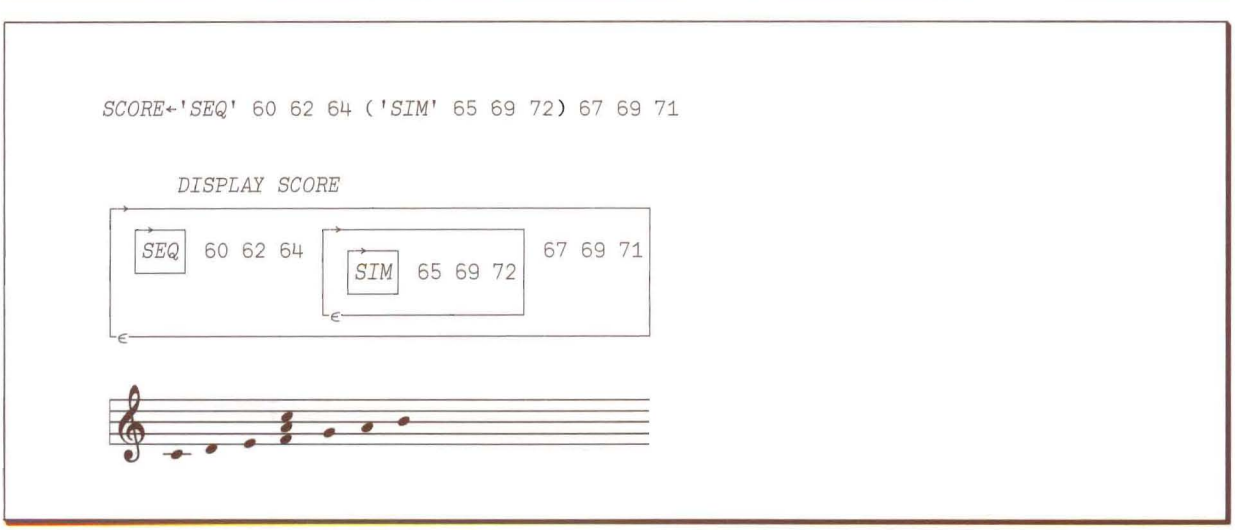
Figure 9 shows a representation of a simple score using the Smoliar model, and implemented in APL2.

One fundamental limitation of this model is that there is no way to explicitly indicate precise durations of time—only order relations between events are expressed. Nonetheless, it is a powerful abstraction for modeling harmonic and tonal structure. By creating a hierarchy with these kinds of nestings at many levels, one can model an entire composition, yet have access to its parts at all levels of the structure. The hierarchies are musically significant because they model how we actually parse real musical events, and how these events group into larger events.

Conclusion

The ideal of a shared notation that can be read by both humans and machines can only be realized if the notation is close enough to human thought to be practical. Our minds must rise above the ancillary details of computation and even implementation, so we can be free to contemplate complex concepts more clearly. Music in particular requires this freedom because musical structure itself is so complex. Even simple-sounding passages can re-

Figure 9 A Smoliar model score



veal surprising complexity when analyzed. The examples in this paper can attest to this—so many numbers to describe such simple fragments of music. One can imagine what would be required to describe a symphony.

APL2 provides a solid conceptual foundation for information processing. Suddenly we have control by attributes. We can specify parts or aspects of the music that we wish to examine or modify. And most important of all, we can create new schemes for classifying these structures, so that the foundation, though solid, remains flexible enough to follow in any direction.

Composers have long employed complex notation systems, attempting to capture the essence of what they wish to express. Theorists seek to understand how we hear music, and attempt to make maps of possible musical spaces. Both require a language in which new languages can be easily defined. In the computer age, APL2 seems to be an important evolutionary step along this path.

Cited references

1. W. Berry, *Structural Functions in Music*, Dover Publications Inc., New York (1987).
2. P. Smith, *A Programming Language for Thoughts and Dreams*, Technical Report 77.0175, IBM Information Products Division, P.O. Box 1900, Boulder, CO 80301-9191 (1986).
3. R. Lafore, *Microsoft C Programming for the PC*, Howard W. Sams & Company, Carmel, IN (1990), pp. 21, 327.
4. J. Backus, "Can Programming Be Liberated from the Von Neumann Style?—A Functional Style and Its Algebra of Programs," *Communications of the ACM* **21**, No. 8, 613–641 (1978).
5. P. Benkard, "Rank vs. Depth for Array Partitioning," *APL84 Conference Proceedings*, *APL Quote Quad* **14**, No. 4, 33–39, ACM, New York (June 1984).
6. J. Brown, *The Principles of APL2*, Technical Report 03.247, IBM General Products Division, 5600 Cottle Road, San Jose, CA 95193 (1984).
7. L. Van Noorden, "Two-Channel Pitch Perception," *Music, Mind, and Brain*, Manfred Clynes, Editor, Plenum Press, New York and London (1982), pp. 251–269.
8. M. Babbitt, "Twelve-Tone Invariants as Compositional Determinants," *The Musical Quarterly* **46**, No. 2, 245–249 (1960).
9. *Musical Instrument Digital Interface (MIDI) Specification 1.0*, The International MIDI Association (IMA), Sun Valley, CA (August 1983).
10. F. R. Moore, *Elements of Computer Music*, University of California at San Diego, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990), pp. 12–14.
11. S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, Inc., New York (1980), p. 21.
12. E. S. Friis and S. Jordan, "Musical Syntactic and Semantic Structures in APL2," *APL90 Conference Proceedings*, *APL Quote Quad* **20**, No. 4, 130–139, ACM, New York (August 1990).
13. S. Smoliar, "A Computer Aid for Schenkerian Analysis," *Computer Music Journal* **4**, No. 2 (1980). The MIT Press, Cambridge, MA, and London, England, pp. 41–59.
14. H. Schenker, *Der Freie Satz*, Universal Edition, Vienna, Austria (1935); Ernst Oster, Translator, Longman, New York (1979).
15. N. Chomsky, *Syntactic Structures*, Mouton, The Hague, Netherlands (1957).

Accepted for publication June 11, 1991.

Stanley Jordan *163 3rd Avenue, Suite 143, New York, New York 10003.* Stanley Jordan received his B.A. in music from Princeton University, where he studied music theory and composition with Milton Babbitt and computer music with Paul Lansky. He is a composer, guitarist, and a recording artist with Arista Records. In 1985, his Blue Note album, "Magic Touch" was *Billboard* magazine's number one jazz album for 51 weeks. Mr. Jordan is widely acclaimed as the foremost innovator of the "Touch Technique," or "Tapping Technique," which allows one guitarist to sound like two or three. He has been developing computer music applications in APL since 1978.

Erik S. Friis *Matrix Development Corporation, Suite B, 19 Shadow Lane, Montvale, New Jersey 07645.* Mr. Friis graduated from Rensselaer Polytechnic Institute in 1983 with a B.S. in computer science cum laude. He joined IBM in 1983 and was involved in software design and development for eight years. In 1991 he left IBM to start the Matrix Development Corporation, a software company. He is the author or coauthor of several papers published by SIGAPL of ACM and has authored several IBM Technical Reports. He presented papers at the APL89 and APL90 conferences, and he was an invited speaker, along with Stanley Jordan, at the APL91 conference held at Stanford University.

Reprint Order No. G321-5450.

Verification of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator

by A. Aharon
A. Bar-David
B. Dorfman
E. Gofman
M. Leibowitz
V. Schwartzburd

Verification of a computer that implements a new architecture is especially difficult since no approved functional test cases are available. The logic design of the IBM RISC System/6000™ was verified mainly by a specially developed random test program generator (RTPG), which was used from the early stages of the design until its successful completion. APL was chosen for the RISC System/6000 RTPG implementation after considering the suitability of this programming language for modeling computer architectures, the very tight schedule, and the highly changeable environment in which RTPG would operate.

The ultimate goal of design verification is to ensure equivalence between a design and its functional specification. Strictly speaking, we can say that this goal can be achieved by exhaustive simulation or formal proof of correctness. The exhaustive simulation, in which all possible combinations of all inputs and memory elements of the design should be applied, can be done only for very small designs. Also, the state of the art of the formal techniques and the complexity of designs and specifications, usually written in English, do not allow utilization of the formal techniques in most industrial applications.[1] Despite significant progress achieved in recent years in formal verification, it has been pointed out that formal verification is not intended to replace simulation completely and that simulation is presently the major tool for the (partial) validation of hardware designs.[2]

In practical applications only a relatively small subset (as compared to the exhaustive set) of selected test cases is simulated. The challenge, then, is to create a subset that provides high confidence in the correctness of the design. We discuss how biasing techniques, combined with the dynamic approach to random test program generation, help to solve this problem.

This paper describes the concepts behind and the implementation of the IBM RISC System/6000* random test program generator (RTPG) developed to assist in the interactive creation of the adequate subset, as well as to automatically produce a vast

number of test programs for the comprehensive verification of the design.

At the moment the design was launched, no functional test cases existed for RISC System/6000 architecture. It was obvious that the traditional way of writing test cases could not provide the required level of confidence in the design. The design verification methodology developed at the IBM Haifa Research Group (HRG)[3,4] had already been successfully applied to several smaller designs such as floating-point units and a microcontroller. It was decided to adopt this approach for verification of the RISC System/6000 computer system.

APL was chosen as the programming language for the RISC System/6000 RTPG after considering the suitability of this language for modeling computer architectures, the very tight design schedule, and the highly changeable environment in which RTPG would operate. Originally the RISC System/6000 RTPG was developed in VS APL on the virtual machine (VM) operating system. It was later "migrated" to APL2 on the same system. It is currently being used for verification of follow-on products and is running mainly in batch mode on a cluster of over 30 RISC System/6000 machines, using IBM's APL2/6000.

The second section of the paper discusses some aspects of processor verification and describes a test program format suitable for this purpose. The subsequent section presents the main RTPG concepts and ways to realize them. The RTPG structure and the basic operation modes are described in the fourth section. Highlights, conclusions, and results of the RTPG experience are summarized in the last section.

The nature of processor verification

The RISC System/6000 RTPG and its predecessors. Logic verification of VLSI (very large-scale integrated) designs has always been an intrinsic part of the design process; however, the complexity of verification grows much faster than the complexity of designs. The problem is widely recognized in the case of microprocessors, as they present the leading edge of single-chip design complexity. Verification of microprocessors is considered to be a bottleneck of the entire design process, with crucial impact on the schedule for delivery of new systems.[5] An automated approach is essential for verification of a system that consists of several VLSI chips including a processor, a floating-point unit, a storage control unit, and caches.

In most applications, a test case for a processor is a program written in assembly language. The main goal of the tool called *RTPG*,[6] which is actually a *dynamic biased pseudo-random test program generator*, is to make the test program generation process more productive, comprehensive, and efficient.

A random approach to automatic test generation has proved to be successful.

A random approach to automatic test generation for software[7] and hardware[8] verification has proved to be successful. It was applied to the verification of selected design units such as a floating-point unit[9] and even a complete processor,[8] but very strong restrictions were imposed on the generated test programs. As a result of those restrictions, many parts of the design could not be accessed and, thus, could not be verified.

For some designs, such as a floating-point unit, the main part of the verification task can be fulfilled by programs that consist of only one instruction. The generation of such programs is relatively straightforward: The generator (or the user) selects an instruction and the required controls, generates the operands randomly (or provides them), and then invokes a reference model of the design to get the expected results. The generation of multiple instruction test programs is much more complicated, especially when such features as program control instructions, interrupts, and address translation are to be verified. There are approaches[8,10] that present a way to generate multi-instruction test programs. They are based on creating special tables of operands, and each instruction may select operands only from the relevant tables. Although much more productive than manual test writing, these approaches have drawbacks. The tables used must ensure that the generated test programs are worthwhile, that they would create the required instruction stream, and that they would not quickly end up with an

interrupt. These conditions imply use of the utmost caution in creating the tables and make this task very cumbersome. The generated test programs are relatively simple, and again, must obey many restrictions. For example:

- Some instructions are always preceded by specially inserted instructions, e.g., for initialization of base registers to get the allowed memory addresses. Thus, some sequences of the instructions can never be generated.
- For the same reason, a register may not be used as a source for different types of activity, such as an operand in an arithmetic instruction and a base register for addressing.
- To avoid creating endless loops, only branch forward instructions are generated. In branch conditional instructions, where it is not known *a priori* whether the branch is taken or not, instructions for both possible paths must be generated. As a result, the generation of test cases with many branch conditional instructions is quite difficult.

Such an approach to test generation may be classified as a static one, since the test programs are assembled first and executed afterwards. There is no relation between the intermediate machine states during execution of the test program and the test generation process. In RTPG the test generation is interleaved with the execution of every instruction as soon as it is generated. This dynamic nature of RTPG allows us to overcome drawbacks of the static approaches.

Because RTPG makes it easy to write test programs, it encourages the logic designer to create appropriate test programs while the logic design is still fresh in the designer's mind, whether these programs can be simulated at that early stage or not. Thus, the design verification is naturally integrated into the design process.

There are two challenges in having a test program generator ready at the early stages of design. The first is simply the time required to implement the test program generator. The second is a requirement for high flexibility. Frequent changes are required to the generator because the architecture specification is often very much in flux at this time and because implementation-specific details of the test programs are decided as the design progresses.

These requirements are two of the reasons that APL was chosen for the RISC System/6000 RTPG implementation. APL provides quicker implementation than many other languages. It is also easy to modify to meet changing requirements as well as to handle various designers' requests. Another reason is the special suitability of APL for describing and modeling computer architectures. From its very beginning, APL was used for this purpose. Iverson's original book[11] contained a description of the IBM 7090 machine, and in 1964 the complete System/360* was formally described in APL.[12] All Boolean and relational functions are supported, and these functions provide very efficient bit-per-bit execution for bit arrays of any length. The language has the ability to individually address each bit in an array. It is often necessary to work with bit fields and subfields within instruction or data words, including double-precision floating-point data which are 64 bits long in the RISC System/6000. APL allows the needed fields to be easily split out, whereas many languages do not support bit operations at all (and especially not in more than 32-bit words). Because bit operations are so common, RTPG keeps values for all of the instruction and data words in Boolean form. Since APL stores a Boolean value as a single bit in the host processor storage, there is no penalty for keeping data in this convenient form. The APL rotate function together with the selection functions (like **take** and **drop**) are natural for implementing bit-shifting operations that are required in any computer processor model and are especially powerful in the IBM RISC System/6000 architecture. For example, the result of a Shift Right Algebraic Immediate (SRAI) instruction[13] is calculated by the following concise expression:

```
GPR[RA;]←32↑(SH/GPR[RS;0]),GPR[RS;]
```

and the Carry bit (CA) is:

```
XER[2]←GPR[RS;0]∧∨/(-SH)↑GPR[RS;]
```

Here *SH* is the shift amount, *RS* and *RA* are the numbers of the general-purpose registers involved in the instruction, and the second bit of *XER* (fixed-point exception register) contains CA. A description of the same instruction in any other programming language would be much longer and less easily understood. Note that the description of this instruction in English takes 10 lines in the architecture document.

Finally, although RTPG was initially planned to run on an IBM 3090-type processor (under VM), it was recognized early-on that it would also be necessary to run RTPG on workstation platforms. In fact, RTPG is now running under IBM's APL2/6000 on the very platform that it helped to verify. The transfer of the RTPG APL code from APL2 on the VM operating system to APL2/6000 on the Advanced Interactive Executive* (AIX*) operating system was trivial. The only change required was to the four file I/O programs and to the display screen programs. As mentioned earlier, RTPG is now running on a cluster of over 30 IBM RISC System/6000 processors to do the work of verifying new processors under development for the RISC System/6000 family of computers.

Test programs for processor verification. The most natural way of processor verification is to run assembly programs through the design model and to compare the simulated results with the expected ones. Usually the test programs are written as self-checking programs that return only a "go/no-go" flag. This concept is simple; however, its usage faces difficulties since:

- It can be used only when the design model is at an advanced stage, or at least when load and compare instructions are implemented.
- The test programs should obey the restrictions imposed by the supervisor that runs them.
- It requires more simulation cycles (running time) because of additional load and compare instructions that are simulated.

The RTPG approach is different. A test program generated by RTPG consists of three parts:

1. *Initial state* defines the contents of all registers, control flags, tables, caches, and memory locations (called "facilities") that influence, explicitly or implicitly, the execution of a test program. The instruction pointer (IP) register provided in this part contains the program initial address.
2. *Instructions* are given as the contents of caches or memory locations or both. The instructions part may be included in the initial state but is separated for better readability.
3. *Expected results* present the final state of all facilities that were changed during the test. The user may request that the final state of additional facilities also be included in the expected results. The IP register provides the test program breakpoint.

The collection of these three parts is referred to as a *test program* in this paper. Such test programs are self-contained: they include all information required for their independent and completely predictable execution. This feature enables them to freely migrate between test libraries and to be executed in any order.

A small test program generated by RTPG for the IBM RISC System/6000 processor is shown in Figure 1. As usual, asterisk "cards" (or lines) are used for comments. Comments may also be included in any line after the required data. The header (H) card contains the test number and indicates the beginning of the test. The register (R) cards specify register names and initial values. The instruction (I) and data (D) cards provide memory addresses and their contents. The IP values (both the initial value and the result) are given as effective addresses, i.e., before any address translation is performed. All other addresses are given as real memory addresses. The I and D cards are essentially the same and have different tags for readability only.

In addition to the data required for the processing, an RTPG-generated test program contains the following information:

- User comments to record the purpose for which the test has been created
- Corresponding assembly code (in the I cards) for readability
- Calculated effective address of data and target instructions, included as comments in I cards of load, store, and branch instructions
- The translation path of each address when a test program is running in address translation mode (not demonstrated here for reasons of clarity)
- The initial value of the Random Link (□*RL*) used to create the test and other control parameters required for regeneration of the test program (Only a few of these are shown in Figure 1.)
- Hooks for handling the program in test libraries

When requested, intermediate results of each instruction are included as comments in the *instructions* part of a test (the debug mode described in the fourth section).

Realization of the RTPG principles

RTPG realizes the dynamic approach to test generation in the following way. A test program is built step by step (instruction by instruction). Each step

Figure 1 An RTPG-generated test program for the RISC System/6000

```
* --------------------  RISC System/6000 RTPG  -------------------
H 10000;
* Created by: UserId                                Mar 28 12:38:17 1990
* Title   :  A simple test program for RTPG paper
* Comment:  Add, Load, Branch, and Store instructions
* Number of tests: 1;   Instructions in test: 4;
* Instructions: a  lx b sth
* Seed: 228656141;  FN: example;  Instr. order: f;  New_Reg: y;
*
* --------------------  Initialization  ------------------------
*
R IP        00010000
R R1        03642998
R R8        0000000F
R R10       1E12115F
R R22       0129DFFF
R R30       800000BA
R MSR       00008000
R CR        8CC048C8
R XER       2000CD45
D 0129DFFC 4E74570E
D 03640B90 7D280411
*
* ---------------------  Assembly Program  ----------------------
*
I 00010000 7C48F415  ao.       R2,R8,R30
I 00010004 7CE0B02E  lx        R7,R0,R22          * E/A 0129DFFF
I 00010008 49BBB904  b         *+29079812         * T/A 01BCB90C
I 01BCB90C B141E1F8  sth       R10,X'E1F8'(R1)    * E/A 03640B90
*
* ---------------------  Expected Results  ----------------------
*
R IP        01BCB910
R R2        800000C9
R R7        4E74570E
R MSR       00008000
R CR        8CC048C8
R XER       0000CD45
D 0129DFFC 4E74570E
D 03640B90 115F0411
END
```

consists of two main stages: a generation stage and an execution stage. At the generation stage a new instruction is chosen, and the required facilities are initialized. The execution stage is then invoked to execute the instruction and to update the affected facilities.

Dynamic test generation. The generation stage starts by inserting the operation code into the instruction word and establishing the rest of the instruction fields. At any point in the process each facility may be either free, which means that no value has been assigned to it yet, or have a value, in which case one is assigned to it by the initialization part of the process or by the execution of previous instructions. All facilities that influence the execution of the generated instruction are inspected, and those that are free are initialized. This principle works regardless of the complexity of a generated instruction and the initialization that it requires. For example, a single store instruction, besides initialization of data, base, and offset registers, may require initialization of an entire address translation path.

As soon as the instruction and all of the associated facilities are defined, the instruction is executed and all facilities that are changed during the exe-

cution are updated. Therefore, at the beginning of the generation of the next instruction, RTPG has the exact information about the current state of all facilities in the system. This information allows RTPG to:

- Select an instruction and its fields to gain the best effect from the execution (various biasing strategies help to achieve this goal)
- Bias data and operands, depending on the instruction being generated
- Reject the instruction if its execution would render the test program invalid
- Include any number of branch instructions in the test
- Control eventual interrupts
- Protect the required areas of memory and certain registers from being used by the generated test
- Define, on the fly, all required entries in the system tables (such as the page frame table)

A trace of the most recently updated facilities is also available and is used for implementation of some useful RTPG options.

Biasing. The generation of test programs is biased in order to increase the probable occurrence of events that otherwise have very low chances of being created. Biasing is both the strong point and the vulnerable point of RTPG. Strong, because it allows the generation of test programs with the required features. Vulnerable, because the selection of biasing strategies cannot be completely formalized and depends on the experience of the RTPG developer and that person's knowledge of the design. The goal of the biasing is not the creation of unique or very rare situations, but rather is to direct the generation process toward selected design areas so that most of the events in these areas are tested when the number of generated test programs is reasonably large.

The biasing functions are employed in the process of selecting instructions, instruction fields, registers, addresses, data, and other components that construct the test program. The starting set of the RTPG biasing strategies is derived from the architecture. Each instruction or process (such as interrupt action or address translation) specified in the architecture is represented by a block diagram composed of decision and execution blocks. In every decision block the data affecting the decision are selected in such a way that the subsequent blocks are entered with user-specified or RTPG-controlled probability. However, in multi-instruction test programs it is not always possible to get the required data. Let us say that the user asked for a 10 percent probability of floating-point overflow, and in the current instruction the decision was made to create it. If all floating-point registers already have values assigned by previous instructions, it may happen that no pair of registers will produce an overflow. Thus, in the generated test cases the actual probability of overflow might be less than the requested one.

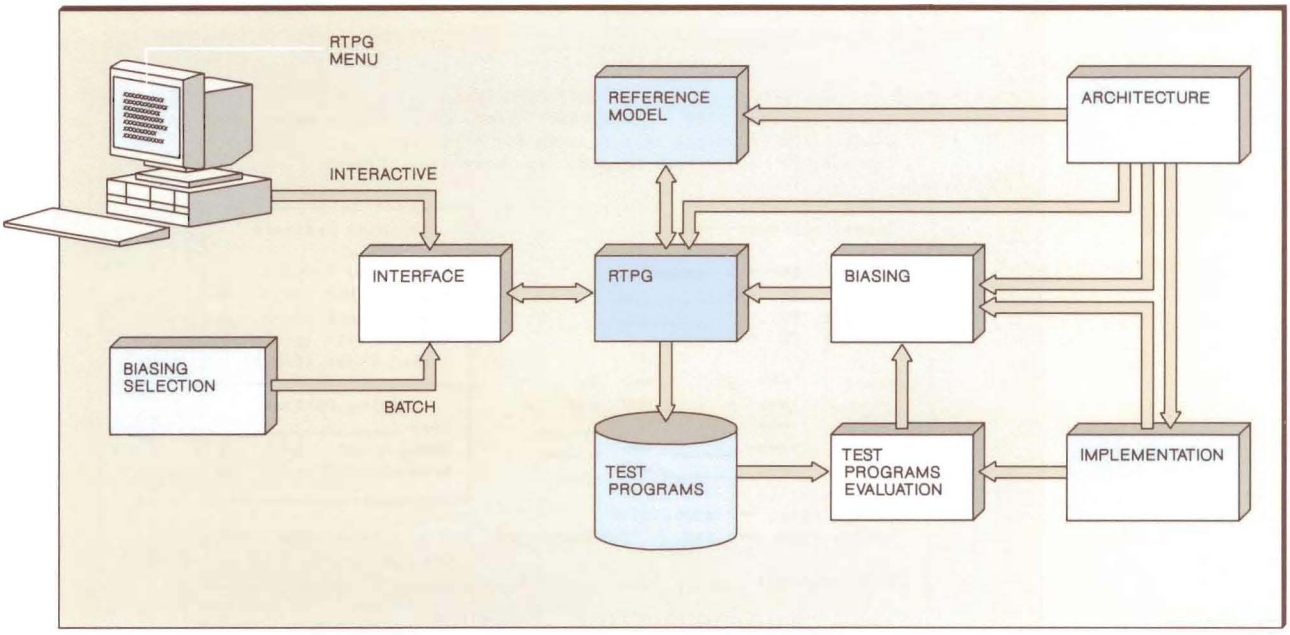
To some extent RTPG is a system that gathers into its biasing strategies all of the experience gained during the verification of several processor designs.

RTPG gathers into its biasing strategies all of the experience gained during verification of several processor designs.

As an example, consider the strategy of register selection, which is very important, especially for a RISC-type architecture where a large number of general-purpose registers is available and as many as three or four registers may be used in one instruction. The RISC System/6000 RTPG allows the selection of any one of the following three strategies:

1. Selection of free registers only. Here RTPG has complete freedom in biasing the instruction operands. Also, the result of each instruction will never be overwritten by the actions of subsequent instructions. Only relatively short test programs can be generated when this option is chosen.
2. Random selection (the default strategy). A register is selected randomly with biasing toward:
   - Increasing the probability to use the same register more than once in an instruction.
   - Preventing usage of a register as a target if it has been a target during its previous usage. This feature increases the test program observability, i.e., the probability to propagate any intermediate errors to the observable expected results.

Figure 2 The RTPG environment



3. High probability to select the target register of the previous instruction as a source or target of the current instruction. This option is useful in the verification of the register bypass logic as well as in the verification of synchronization between instructions when multiple instructions are issued and executed concurrently.

The starting set of biasing strategies is revised based on test coverage analysis of the generated test programs on both the RTPG reference model and the design models.[4] Coverage evaluation helps to detect and remove "holes" in the biasing. The final set of strategies ensures that there is a generation process with a reasonable probability of covering every architectural feature and every design block.

RTPG supports two biasing levels: local and global. The local biasing is involved in selecting immediate fields of instructions and selecting data for the operands. Many local biasing functions, specific for every class of instruction, are implemented in RTPG. For example, the generation of operands for add class instructions ensures a high probability of getting long chains of carries. In a "count leading zeros" instruction the biasing ensures the creation of operands with equal probabilities for any number of leading zeros. Some more sophisticated local biasing functions are implemented in more complicated cases, e.g., floating-point instructions.

Examples of global biasing control parameters that have a primary effect on the generation process are:

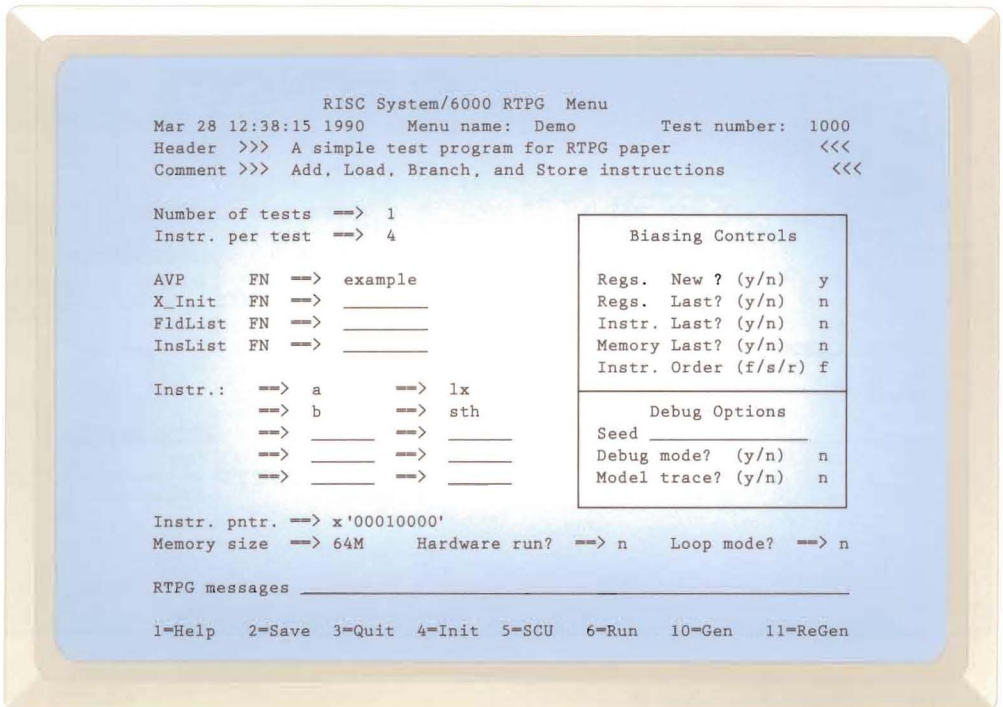
- Instruction selection strategy
- Initial value of the machine state register (MSR)
- Strategy for selecting general-purpose registers
- Memory areas that the test program is allowed to use

Each global and most of the local biasing strategies may be specified by the user. They are selected randomly by RTPG if not provided.

RTPG structure and basic operation modes

Environment. The RTPG design verification environment is shown in Figure 2. The *interface* and *biasing* blocks are actually parts of RTPG but are shown separately because of their connections to the external world. RTPG includes a *reference model*, a high-level architectural model of the processor to be tested. The lighter-shaded lines indicate flow of

Figure 3 The main RTPG menu



information with manual work involved in the process, and the darker lines indicate automatic flow of data between the units.

The architecture specification document is the primary source of information for the RTPG developers, and most of its features are embodied in RTPG. RTPG has to know all of the instruction format, and for each instruction, all of the parameters that influence its execution. This information is required for generating the instruction fields and for checking that all necessary facilities were defined before the execution of the instruction. RTPG has to keep a record of all facilities changed during instruction execution. The final state of all of the changed facilities provides the expected results.

The architecture may leave the handling of certain situations to the implementation. For example, unaligned storage access may cause an alignment interrupt in some implementations and may be handled by hardware in others. All such situations are handled in RTPG so that the test program that is created is correct for the implementation being tested.

User interface. The RTPG *user interface* includes several screens that allow the user to define the initial state of the processor and to control the test program generation process. The main screen used to generate the test program of Figure 1 is shown in Figure 3.

All screen parameters are optional, and if not specified, the default values are used (e.g., the default for InsList contains all instructions). The interface provides a way for documenting the generated test programs, selecting biasing strategies, initializing instruction fields, registers, and memory, and executing existing test programs.

The user may initialize any of the required facilities within an X_Init file. The file has the usual test program format. Thus, a prototype of a test program may first be created by RTPG and thereafter used for initialization and generation of many new test programs on top of the prototype. The X_Init file may also contain blocks of instructions and data that become parts of the test program. This feature is useful for including interrupt handler routines in the generated test programs.

The Debug Mode is a powerful by-product of the dynamic nature of RTPG. Including it in RTPG is almost free since RTPG already scans all changed facilities after every executed instruction. When this option is employed, the expected results of each intermediate instruction are included in the generated test program. This option is very useful in locating problems when a test fails. However, it requires much more space for storing the test programs.

RTPG offers two modes of operation.

RTPG offers two modes of operation: the generation mode (Gen) and the execution mode (Run). In the first mode, RTPG is used to generate one test file per invocation. The file contains the requested number of test programs, each program generated according to the control parameters specified on the screens. The batch version of the Gen mode is used for mass production where a large number of test programs is created for predefined sets of initial conditions. Such generation is performed as overnight runs or during weekends. Since the "porting" of RTPG to the IBM RISC System/6000 platform, RTPG runs as a background process concurrently on many RISC System/6000 machines connected in a local area network. The programs generated on the workstations as well as the programs generated on the VM host machines are submitted automatically to various simulators connected to the same local area network. Tests that do not expose any design problems are discarded.

In Run mode RTPG executes an existing file of test programs and returns it, including correct expected results. The original file may or may not include expected results. If they are provided, they are compared with the expected results created by RTPG, and any discrepancies are reported. Run mode is used to define the expected results for manually written test programs. In case of changes in the architecture, it is used to confirm or update the expected results of programs imported from other sources or generated previously by RTPG. Another use of Run mode is to rerun an existing test program in Debug Mode. This use is done frequently when a test fails and when it was originally generated with the Debug Mode turned off.

Test coverage evaluation on the architecture level. The quality of verification is improved significantly when there is a means to estimate test coverage on both architecture and implementation levels. In regard to RTPG, the results of coverage analysis provide feedback for improving the biasing functions. Also, the coverage analysis on the architecture level is used in the preparation of a relatively small subset of tests that include test programs for every architectural feature. In the RISC System/6000 RTPG the high-level reference model was implemented within RTPG. The interpretive nature of APL considerably facilitated the implementation of some coverage analysis techniques, including techniques that require fault injection.

One of the simplest coverage techniques is ensuring that each line of the code has been executed at least once. A special function analyzes the character representation of an APL function and splits each labeled line into two lines. The first one contains the label and an assignment statement that sets the corresponding bit in a trace vector associated with this function. The second line created by the split contains the APL statement that was on the line before the split (but without the label). Assignments of the bits of the trace vector are also inserted after each statement with an APL right arrow (**branch**). A bucket of test programs is then executed (using the Run mode) on the "trace-modified" RTPG. Zero values in the trace vector indicate blocks that were not reached.

Another coverage technique called "skip mutation,"[14] which requires injection of faults into the code and thus provides much higher confidence in the generated test programs, may also be easily implemented. Skip mutation means that one line of the analyzed function is not executed. To make this technique more sensitive, an original APL function may be replaced by its more detailed version. Skip mutation is performed by another function that takes the character representation of an APL function to be checked and precedes the required line by the comment symbol "⍝". The information from the previous step (line coverage) is used to select only those test programs that pass through the

skipped line. The procedure is repeated for each line in the function.

The coverage analysis is done automatically as soon as both the function to be analyzed and the test file are specified. However, because of performance considerations, only functions for which a low coverage is suspected are analyzed. In the RISC System/6000 RTPG environment, only functions that implement the floating-point unit were analyzed by both techniques.

RTPG implementation. The RISC System/6000 RTPG is implemented as a single workspace which is able to create test programs of up to several thousand instructions on a six- to eight-megabyte virtual machine. The user interface was written in REXX to simplify some Conversational Monitor System (CMS) file-related checking that was not so easy to implement in VS APL. The RTPG functions may be grouped into:

- Utility and service functions
- Functions for instruction execution
- Biasing functions
- Simulation of interrupts
- Address translation

The service functions prepare the initial machine state, manage instruction selection, prevent the creation of endless loops in the generated test programs, and mask undefined results. These functions also handle the Run option, including comparing the actual results with the expected ones that are provided in the original test program.

Each RISC System/6000 instruction has a corresponding APL function with the same name that operates on the architectural facilities (defined as global variables) to perform the behavior of the instruction. Each "instruction function" is partitioned into a biasing section and an execution section which are used as required for biasing and setup or reference model operation. As soon as the instruction to be generated is selected, the required APL function is invoked by "executing" the character representation of the instruction mnemonic. Thus, when a new instruction is added to the system, or in case of a change in the instruction mnemonic or behavior, only one function must be added or changed. A rich set of utility functions that perform many of the required common tasks, such as incrementing the instruction pointer or adding two register values, facilitates writing of the "instruction" functions.

The register arrays of the processor are modeled as APL Boolean matrices. The memory is modeled as another Boolean matrix with a companion address vector that maps processor memory addresses to APL matrix indices.

The RISC System/6000 RTPG is implemented as a single workspace.

When the project was started, very little reusable RTPG software existed from previous projects (although the concepts were well understood). At any moment no more than four persons were working on RTPG. One of them was dedicated to the user interface written in REXX, and one was involved only part time in RTPG development. In less than four months the first version of RTPG, which supported almost all branch and fixed-point instructions, was given to the designers. Floating-point instructions were delivered a month later. From that time only two people on average were involved in RTPG development, working on the storage control unit, on cache modeling, on imbedding architecture changes, and on implementation-dependent features. They also supported RTPG in the field, responding to numerous designers' requests. This activity was completed exactly one year after starting the project, and since then, only one person is involved in RTPG support and enhancements. This person's responsibility includes porting RTPG to APL2/6000 and upgrading it to support follow-on designs.

Concluding remarks

We described a comprehensive procedure for biased random test program generation and the RTPG implementation of the approach. This approach has been adopted as a main technique in the design verification process of several IBM designs. The designs varied from floating-point coprocessors to the complete complex of the IBM RISC System/6000

computer. In all cases the corresponding VLSI products came out fully functional on the first pass. In the case of the RISC System/6000, once the final design was completed, no new bugs were found.

RTPG accompanies the design process from its very early stages through its successful completion. At the beginning of the process, RTPG is used by the designers to generate simple test programs directed toward recently developed logic. This use results in a significantly lower error detection rate at the advanced stages of the design. At the system level, RTPG is used mainly in batch mode, where a significant volume of test programs, some of them consisting of up to several thousand instructions, is generated and simulated on the design model. The employed biasing strategies have evolved, based on requests coming from the designers and the feedback from the coverage analysis.

The RTPG development effort is not considered to be negligible. However, it is incomparable with the amount of resources required to achieve similar verification quality with manually written or purely randomly generated test programs. At the moment, RTPG is notably tailored to the architecture it serves. Nevertheless, existing RTPGs provide a good groundwork for the development of new ones, even when the architectures are different.

Choosing APL for the implementation of RTPG allowed us to provide this tool to the designers on a timely basis, and it also allowed us to keep up with many changes and modifications to RTPG necessitated by the novelty of the approach and also by frequent changes in the architecture at that time. Shoulder-to-shoulder work with the designers contributed to the success of the tool but required instant response to their requests. Again, APL, with no compilation and linkage overhead, allowed us to quickly respond to these requests. The interpretive nature of APL was used to implement some test coverage evaluation techniques that work on the architecture level directly in RTPG.

In the beginning, RTPG was used mainly in the interactive mode. With a capability to generate 20 to 30 instructions per second, it provided fairly good response time to the users. The design model simulator running on the VM host machine was slower. Porting of the simulator to the RISC System/6000 platform and the use of hardware assist for the simulation required more and more test programs to feed all available simulators. Moving RTPG to APL2/6000 solved the problem of limited available computer time, and now RTPG, running on a cluster of RISC System/6000 machines, is able to produce the required number of test programs.

Acknowledgments

The authors deeply appreciate the help of their colleagues at the VLSI testing and verification group in HRG, especially Raanan Gewirtzman and Yossi Malka for their work on test coverage evaluation. The authors gratefully acknowledge cooperation with IBM design and simulation teams in Essonnes, Burlington, Boca Raton, and Austin that enabled us to develop this verification methodology and to experiment with it on their designs. Special recognition goes to Israel Berger (HRG) for his innovative contribution to the methodology and his supervision and support, together with Yoav Medan (HRG) and Jerry Long (Austin), during the development process of the RISC System/6000 RTPG.

Cited references

1. P. Camurati and P. Prinetto, "Formal Verification of Hardware Correctness: Introduction and Survey of Current Research," *Computer* **21**, No. 7, 8–19 (July 1988).
2. *Formal Verification of Hardware Design*, M. Yoeli, Editor, IEEE Computer Society Press, Los Alamitos, CA (1990).
3. A. Aharon, A. Bar-David, E. Gofman, M. Leibowitz, and V. Schwartzburd, *RTPG—A Dynamic Biased Pseudo-Random Test Program Generator for Processor Verification*, Technical Report 88.290, IBM Israel Science and Technology, Technion City, Haifa 32000, Israel (July 1990).
4. A. Aharon, R. Gewirtzman, E. Gofman, and Y. Malka, *Hardware Design Verification with Task Models*, Technical Report 88.289, IBM Israel Science and Technology, Technion City, Haifa 32000, Israel (June 1990).
5. N. Tredennick, "Trends in Commercial VLSI Microprocessor Design," *VLSI CAD Tools and Applications*, W. Fichter and M. Morf, Editors, Kluwer Academic Publishers, Norwell, MA (1987).
6. A. Aharon, A. Bar-David, R. Gewirtzman, E. Gofman, M. Leibowitz, and V. Schwartzburd, *Dynamic Process for the Generation of Biased Pseudo-Random Test Patterns for the Functional Verification of Hardware Designs*, Israel Patent Office, Patent Application No. 94 115 (April 1990).
7. D. L. Bird and C. U. Munoz, "Automatic Generation of Random Self-Checking Test Cases," *IBM Systems Journal* **22**, No. 3, 229–245 (1983).
8. A. S. Tran, R. A. Forsberg, and J. C. Lee, "A VLSI Design Verification Strategy," *IBM Journal of Research and Development* **26**, No. 4, 475–484 (July 1982).
9. P. M. Maurer, "Design Verification of the WE32106 Math Accelerator Unit," *IEEE Design and Test of Computers* **5**, No. 6, 11–21 (June 1988).
10. C. Bellon et al., "Automatic Generation of Microprocessor

Test Programs," *ACM/IEEE 19th Design Automation Conference Proceedings* (June 1982), pp. 566–573.
11. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962).
12. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Systems Journal* **3**, Nos. 2 and 3, 198–261 (1964).
13. *POWER Processor Architecture*, IBM Corporation, Advanced Workstation Division, 11400 Burnet Road, Austin, TX 78758 (1990).
14. A. Aharon, I. Berger, E. Gofman, and M. Yoeli, "Testing a Microprogrammed Control Unit," *VLSI and Computers, COMPEURO Conference Proceedings*, Hamburg, Germany (May 1987), pp. 390–393.

Accepted for publication June 20, 1991.

Aharon Aharon *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Aharon joined IBM at the Haifa Research Group (HRG) in 1983 and since 1989 has been the manager of the VLSI Testing and Design Verification Group. From the time he joined the HRG he has been involved in developing a methodology for logic verification of hardware designs and applying it to various designs within IBM, among them the IBM RISC System/6000 for which he received an IBM Outstanding Technical Achievement Award. Mr. Aharon received his B.Sc. degree in 1981 in computer engineering and his M.Sc. degree in electrical engineering in 1983 from the Technion, Israel Institute of Technology. Since 1984 he has been an Adjunct Teaching Associate in the Electrical Engineering Department at the Technion. Mr. Aharon is a coauthor of several papers and technical reports. He also wrote several teaching books published by the Open University of Israel in the areas of digital systems, logic design, computer organization, and microprocessors.

Ayal Bar-David *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Bar-David joined IBM at the Haifa Research Group (HRG) in 1983 and has been involved in the development of tools for logic verification of hardware designs. He received a general award for his participation in the verification of the IBM RISC System/6000. During 1987–1989 he was visiting at Qualcomm Inc., San Diego, California, working on the design of ASICs for digital communications. Presently he is working on the development of CAD tools for VLSI. Mr. Bar-David received his B.Sc. degree in 1983 and M.Sc. degree in 1986, both in electrical engineering, from the Technion, Israel Institute of Technology.

Barry Dorfman *IBM Advanced Workstations Division, 11400 Burnet Road, Austin, Texas 78758.* Mr. Dorfman is an advisory programmer in the simulation/verification area of the Central Electronics Complex Engineering Center. He joined IBM at Austin, Texas, in 1976 after receiving a B.S.E.E. degree that year from Arizona State University, Tempe, Arizona. He held various assignments in circuit and logic design and received an IBM informal award for his design and APL implementation of an engineering processes system. He worked in the information systems area of the Systems Technology Division where he was responsible for developing several software systems. In 1987 Mr. Dorfman joined the IBM RISC System/6000 team where he works today with processor verification and the RTPG program.

Emanuel Gofman *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Dr. Gofman joined IBM at the Haifa Scientific Center and Research Group in 1977. For his work on projects on the Hydraulic Network Solver (1977–1979) and Computer-Aided System for Scheduling School Time-Tables (1979–1981) he received two awards from the Information Processing Association of Israel, first in Implementations of Science and Technology and second in Data Processing in Administration. Since 1981 he has been involved in developing a methodology for logic verification of hardware designs and application of this methodology to various designs in IBM. He spent 1986–1987 as a visiting staff member in the IBM Independent Business Unit, Austin, working on verification of the IBM RISC System/6000. For this work he received an IBM Outstanding Technical Achievement Award. Dr. Gofman received an M.Sc. degree in applied mathematics from the Moscow State University, USSR, in 1968, and a Ph.D. in technical cybernetics in 1975 from Riga Politechnical Institute, Latvia.

Moshe Leibowitz *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Leibowitz is a research staff member in the Haifa Research Group (HRG) VLSI group. He graduated from the Technion, Israel Institute of Technology with a B.S.E.E. degree (cum laude) in 1975 and received an M.S.E.E. degree from the Technion in 1989. He joined IBM in March 1985 at Haifa, Israel, and took part in and led several projects in VLSI testing and simulation and in physical design. Currently he is on international assignment at IBM Boca Raton working on VLSI synthesis and design verification.

Victor Schwartzburd *IBM Israel Science & Technology Ltd., Technion City, Haifa 32000, Israel.* Mr. Schwartzburd joined IBM at the Haifa Scientific Center in 1984. He has been involved in developing a methodology for logic verification of hardware designs and application of this methodology to various designs in IBM. From 1986 to 1988 he worked on development of an automatic verification system for the IBM RISC System/6000. For this work he received an IBM Outstanding Technical Achievement Award. He also has a patent (with other IBM employees) on the method of automatic verification and testing. He spent 1990 and 1991 as a visiting staff member in the IBM Storage Systems Products Division, Tucson, Arizona, working on testing of the IBM 3990 Control Unit. Mr. Schwartzburd received an M.A. degree in electrical engineering from the Moscow Power Institute, USSR, in 1957.

Reprint Order No. G321-5451.

APL2 as a specification language for statistics

by N. D. Thomson

APL has had a dedicated following for many years among some sections of the academic and industrial statistical communities. One of its greatest strengths is its value as a specification language. Not only can algorithms be described consistently and unambiguously, but also, given an appropriate interpreter, the specifications can be immediately executed. A group of academic and industrial statisticians in the United Kingdom recognized these capabilities and embarked on a project called ASL (APL Statistics Library) with the support of the British APL Association. ASL aims to provide a collection of coherent APL functions for widely used statistical calculations, thereby creating standards for the unambiguous expression of statistical algorithms. A natural consequence of this is that discussions of more complex algorithms and methods can occur without the need to revisit and redefine basic functions and the ways in which they interpret data.

Many statistical algorithms already exist in APL. The APL Statistics Library (ASL) is unique, however, in the way in which it uses APL2 as a specification language for statistical functions, which themselves define a statistical sublanguage with a high degree of consistency and extendability in its naming conventions. This allows users of ASL-based software to predict with greater accuracy the purpose and usage of a function from its name and argument names.

In software engineering, specification languages exist to allow programmers to evaluate programs and their correctness at all levels of detail. APL provides this facility for statistical algorithms but with the important additional property of *executability*. This means that ASL code used for the purpose of specification can be submitted to an APL2 interpreter and executed.

Further, by having algorithms defined at APL source level, potential users are given much greater control over their analyses than they would have using conventional packages. Alternative functions are available to perform operations such as matrix inversion, numerical integration, random number generation, and approximations for functions associated with distributions. Users can substitute their own functions at appropriate points in an algorithmic sequence.

This paper gives examples that (1) describe the philosophy of ASL code and documentation, and (2) illustrate the way in which it provides a medium for discussion of algorithms among statisticians.

ASL structure

ASL is structured into "volumes." The foundation volume is called the Basic Statistics Volume and has two key roles: first, that of specifying a core of algorithms that statistical practice and experience require as basic; and second, that of standardizing the statistical sublanguage, thus giving users of ASL a great general advantage in communicating with each other. Later volumes that cover more specialized areas such as regression, time series, and mul-

Table 1 Coding system for writing functions and naming arguments

Prefix	Algebraic Type	Rank	Suffixes
	b		
ne	n	s	
f	z	v	v
s	(r)	m	m...
	c	a	a
	g		c

Explanation of Codes

ne	nonempty
f	frequency
s	shape
b	Boolean
n	non-negative integer
z	integer
r	real (default)
c	character
g	general, i.e., character or numeric
s	scalar
v	vector
m	matrix
a	array
c	continuous

tivariable analysis can utilize and build on the existing core of algorithms.

The Basic Statistics Volume is divided into a number of sections covering univariable statistics, distribution functions, elementary multivariable statistics, estimation and significance testing, nonparametric statistics, basic time series, analysis of variance, and combinations.

ASL function naming conventions

The function names used in the statistical sublanguage are inflectional, and the general pattern of a function name is

```
<root> <inflection..>
```

where the dots denote the possibility of multiple occurrences.

Sometimes alternative algorithms are given, for example, for generation of random normal variables, which differ in characteristics such as elegance, speed, and space requirements. These are given serial numbers as a further inflection.

Sometimes a function calls a succession of auxiliary functions. This may occur in a set of tightly bound recursive functions such as arise in combinatoric algorithms. The auxiliary functions are named by applying successive Δs as prefixes to the root or root + inflection, so that the full specification of a function name is

```
[Δ..]<root><inflection..><serial number>
```

The largest root-based groups are statistics and distribution functions. Examples of function names are the following:

MEAN	
MEANFM	mean of a frequency matrix
PCTILEFMC	percentile of a frequency matrix—continuous case
NORMQUANT	quantile of normal distribution
FTAIL	probability in right tail of F distribution
PERM	list of permutations of given order
Δ*PERM*	auxiliary to *PERM*
ΔΔ*PERM*	auxiliary Δ*PERM*

The following conventions are used in writing functions and naming arguments:

1. Function results are denoted by *Z*.
2. A left or right argument is specified either by a descriptive name such as *IXSET* standing for "index set," or by a composite "word" made up of not more than four parts, in lowercase letters, which describe the argument type using the coding system illustrated in Table 1. The type fm (frequency matrix) describes the special case of a numeric matrix with two columns, the first of which is to be interpreted as denoting class values in ascending order and the second as a vector of integers giving the number of items belonging to each class.
3. A reasonable degree of abbreviation in naming functions and arguments is employed to avoid excessively long names. Function names are never less than four characters in length.

Examples of functions from the Basic Statistics Volume

The meaning of a statistical function such as "mean" is data-dependent. If it is applied to a vec-

tor, i.e., a sequence of numbers, then the underlying calculation will be different from that performed if it were applied to a matrix and the result defined to mean a sequence of column means.

It has always been a fundamental part of the philosophy of APL to generalize function semantics with regard to data. In this spirit, it is possible to have the same APL expression realize both of the above interpretations of "mean," but the expression is also meaningful if applied to an array of three or more dimensions.

There is, however, a problem associated with this ability to generalize, namely that a data matrix is capable of several different interpretations. For example, each column of the matrix:

```
      DATA
0 1
1 4
2 3
3 2
```

may be regarded as a vector of values of a variable. On the other hand *DATA* can be interpreted as a frequency matrix as described in the previous section—that is, one item with value 0, four items with value 1, and so on. Another interpretation might be two items lying between 0 and 1, four items between 1 and 2, etc., with the items in each class spread uniformly throughout the class width. For example the four items in the second class would be spread to values at 1.125, 1.375, 1.625, and 1.875.

The Basic Statistics Volume provides pairs of functions that deal with the multiple-vector and frequency matrix interpretations. For the mean they are called *MEAN* and *MEANFM* respectively, so that the following results hold:

```
      MEAN DATA
1.5 2.5
      MEANFM DATA
1.6
```

The definition of the root (i.e., noninflected) function for *MEAN* applied to a nonempty array (nea) is:

```
[0]    Z←MEAN nea
[1]    Z←(+⌿nea)÷1⌷⍴nea
```

The symbol ⍴ means the "shape" of the array, e.g., 4 2 in the case of *DATA*. The symbol ⌷ means "index" so 1⌷⍴*DATA* is the first item in 4 2, namely 4.

The symbol ⌿ means take sums along the first axis. That is, if *nea* is a vector, *MEAN* returns the mean of a sequence of numbers; if it is a matrix, it returns a vector of column means. If it is a three-dimensional array, then *MEAN* returns a matrix of the means of items occupying the same positions in the different planes, which is useful in dealing with replications of cross-tabulated data.

The function *MEAN* can readily be generalized to calculate other moments about the origin:

```
[0]    Z←n MOMENT nea
[1]    Z←(+⌿nea*n)÷1⌷⍴nea
```

The only difference is the addition of "*n" where * denotes exponentiation.

```
      2 MOMENT DATA
3.5 7.5
```

The moment about the mean can now be specified as:

```
[0]    Z←n MOMENTM nea
[1]    nea←nea-(⍴nea)⍴MEAN nea
[2]    Z←n MOMENT nea
```

Using basic functions to discuss more advanced ones

The example chosen is that of the jack-knife.[1] Suppose that the reader were required to explain this concept to someone who had not met it before but had reasonable acquaintance with basic statistics. The first step might be to describe the process by which items are withdrawn one at a time from a vector *V* to form ⍴*V* samples each of size (⍴*V*)−1. Four additional symbols are needed, all but the first of which belong to APL2 but not to first-generation APL.

- ⍳ **Index of**—which is the vector of positive integers from 1 to *N*, e.g., ⍳3 is 1 2 3
- ~ **Without**—in the sense that *A*~*B* means the object *A* excluding those items which occur in the object *B*
- ⊂ **Enclose**—which means regard the array to its right as a single object (scalar)
- ¨ **Each**—an operator which directs that the function to its immediate left must be applied to each of the items in the argument on its right

Write n for ρv. The sample construction process for the jack-knife is described by first enclosing ιn so that it can be considered as a single unit, then the individual items of ιn are each excluded in turn. This is described by the following:

```
(⊂ιn)~¨ιn
```

and the process can be made into a function, as follows:

```
[0]   Z←JINDEX n
[1]   Z←(⊂ιn)~¨ιn

      JINDEX 3
 2 3  1 3  1 2
```

The result of *JINDEX* is a set of n vectors each of length $n-1$, which are the index sets that must be applied to v to select the samples required for the jack-knife. The process of selection is described by bracket indexing:

```
[0]   Z←IXSET SELECT v
[1]   Z←v[IXSET]

      2 3 SELECT 27 9 52
 9 52
```

This selection process must be applied for each of the index sets, which leads to the following function:

```
[0]   Z←JSAMPLES v
[1]   Z←(JINDEX ρv)SELECT¨⊂v

      JSAMPLES 27 9 52
 9 52  27 52  27 9
```

The **enclose** which is applied to v is necessary because each of the index sets given by *JINDEX* ρv is applied to the single object v which has to be (notionally) replicated once for each index set.

As an aside, the choice of v rather than *nev* for the argument indicates that the algorithm remains sound (although of trivial interest) in the case where v is an empty vector.

The above development may seem a little tedious because each APL2 symbol and function has been described at some length. To illustrate how rapidly this small learning investment pays off, consider how easy it is now to specify another quantity which arises early in jack-knife theory, namely the jack-knife root mean square:

```
[0]   Z←JACKRMSE nev
[1]   Z←((2 MOMENTM
          MEAN¨JSAMPLES nev)
             ×¯1+ρnev)*0.5
```

Conclusion

This paper has endeavored to argue the case for using APL2 within the professional statistics community as a language for standardizing and specifying algorithms. The paper illustrates how a set of such standard APL2 functions can be used as a basis for reasoning about and extending base algorithms. A group of statisticians in the United Kingdom is actively engaged in continuing to develop this work.

Cited reference

1. B. Efron, *Jack-knife, the Bootstrap and Other Resampling Plans*, Society of Industrial and Applied Mathematics, U.S. (1982).

Accepted for publication July 25, 1991.

Norman D. Thomson *IBM United Kingdom Laboratories, Hursley House, Hursley Park, Winchester, Hampshire S021 2JN, England.* Mr. Thomson joined IBM in 1969. He has earned an M.A. in mathematics at Cambridge and a B.Phil. in statistics and computing at St. Andrews. His interests have been in education and in the application of statistics and simulation in manufacturing. Dr. Thomson has been involved in many projects in collaboration with the IBM United Kingdom plants. He has been an ardent enthusiast for APL and is the author of *APL Programs for the Mathematics Classroom*, Springer-Verlag, 1988.

Reprint Order No. G321-5452.

Advanced applications of APL: logic programming, neural networks, and hypertext

by M. Alfonseca

This paper reviews the work of the author on the application of the APL and APL2 programming languages to logic programming, emulation of neural networks, and the programming of hypertext applications.

The last decade has witnessed the emergence and maturation of a whole set of new fields and techniques in computer science, such as logic programming (which actually started in the 1970s), neural networks, object-oriented programming, genetic algorithms, and a few others. APL (and its successor APL2) remains abreast of the times as a programming language and has demonstrated its capability for all of these exciting new fields.

This paper summarizes the previous work of the author in three of the indicated fields. The first section, on logic programming, describes the design of a logic programming auxiliary processor, capable of performing declarative logic inferences similar to Prolog, that can be invoked and used from an APL workspace. This processor is now a part of the APL2/PC IBM product.

The second section, on neural networks, describes how APL can be used to model, teach, and implement these modern structures which, though descending directly from the perceptrons of the 1960s, have now revived with a new strength and are being applied to new, interesting fields.

Finally, a third section summarizes why APL2 is extremely apt for the development of object-oriented applications and describes in some detail a hypertext application built on these lines.

APL and logic programming

The literature on APL shows that there has been a long-standing discussion about the usefulness of this language for artificial intelligence applications. This usefulness is considered a direct consequence of the great power of the language, the ease of programming with high-order data structures, or the possibility of using a "parallel" approach to solve certain problems. Reference 1 gives more details on the latter.

In particular, the new list structures introduced in the APL2 form of the language[2] provide APL with all of the power of LISP, the classical language for artificial intelligence.[3,4]

Several attempts have been made to build expert systems using only the current power of the language, either with APL or APL2.[5–9] Building an expert system usually requires the implementation of

Table 1 Reserved symbols for AP998

APL Symbol	Standard Symbol	Meaning
←	<-	IF
∧	&	AND
∨	|	OR
~	-	NOT

an inference machine, or some version of logic programming, in APL. The problem here is speed. However, some of the approaches find highly original ways to solve this problem.

A related approach is the emulation of Prolog-like rule-based inferences. Although such emulation has been done more than once,[10,11] this approach is usually too slow, for it boils down to interpreting an interpreter.

A better solution to this problem would be the implementation of a Prolog-like inference processor in a lower-level language in such a way as to be easily accessible from normal APL programs. In this way, powerful hybrid systems could be implemented. Applications built in APL using this "logic auxiliary processor" would gain access to a whole class of new possibilities (logic inferences, "natural-like" language, nonprocedural programming) while at the same time maintaining all of the APL numeric calculation and symbolic manipulation capabilities.

This inference processor is already written and is a part of the APL2/PC product. It is an auxiliary processor, called AP998, accessible from APL2 in the usual way through shared variables, and incorporates a subset of a Prolog-like interpreter.

It has been said that this method is not really an APL solution, since it does not use pure APL programs but instead adds one external program (the auxiliary processor) written in a different language. I think this criticism is unfair, because:

- Auxiliary processors are, and have been for a long time, a part of APL. The fact that they are included in the products proves this assertion.
- APL allows the construction of auxiliary processors in different languages, and this capability is a plus, not a minus, of the language. It is a well-known fact that APL as an interpretive language has a certain impact on performance. The standard solution (avoiding loops in the code) is not always feasible, especially when cascaded results are involved, that is, those processes where the next value to be computed depends on previously computed values. In those cases, it is a great advantage to be able to speed the system up by programming the bottlenecks in a lower-level language. If this can be done in such a way that the resulting auxiliary processor is of general application and can be reused in very different contexts, APL becomes richer and increases its power for future applications.

The remainder of this section describes the logic inference auxiliary processor, AP998.

The logic language. The logic language implemented by AP998 is a subset of Prolog using only infix notation. The lexical elements of the language are the following:

- Words—A word can be defined as any character string not including spaces. Uppercase and lowercase are considered to be equivalent. Examples are:

```
JOHN
IS-FATHER-OF
25
```

- Reserved symbols—Certain symbols have special meaning for AP998 and should not be used outside their context. To be recognized, these reserved symbols must be separated from adjacent words by at least one space. Each meaning can be represented by two different symbols, one of which is easier to represent with the APL keyboard, whereas the other is easier to represent with the standard keyboard. The symbols are shown in Table 1.
- Synonyms—Certain words can be defined as synonyms for the reserved symbols. In this way, many natural languages are recognized by AP998. In English, the synonyms recommended for the symbols are the words indicated in Table 1 under the heading Meaning. Only one synonym may be defined for each meaning at a given time.
- Variables—Any character string starting with the "star" symbol (the asterisk, *) represents a variable. Examples are:

```
*X
*CASE
*1
*
```

The syntactic elements of the language are the following:

- Clauses—They are assertions or negations of dyadic predicates, written in infix notation. They consist of a certain number of words or variables, with a possible negation term in any position. They can also include a plausibility integer. Examples are:

```
JOHN IS MALE
JOHN IS FATHER OF JANE
*1 IS NOT FATHER OF *2
?80 WEATHER IS FINE
```

  The plausibility integers are numbers between zero and 100, zero corresponding to the negation of the assertion, 100 to its certainty, and 50 to its uncertainty. If the plausibility of an assertion is not given, it is assumed to be absolute. If the assertion is affirmatively worded, it is used in that form with a plausibility of 100. If the assertion is negatively worded, its negation is used with a plausibility of zero.

- Rules—Basically the rules are formal logic implications. A <- B is equivalent to A IF B, where A and B are assertive or negative clauses. Rules consist of two parts (premises and conclusion) joined by the IF symbol or its synonym.

A special case rule is the "axiom" or "fact," a rule without premises, that reduces to a single clause. Axioms may be considered as assertions or negations of dyadic predicates written in infix notation. Examples of axioms are:

- JOHN IS MALE—equivalent to the Prolog monadic predicate MALE(JOHN)
- JOHN IS FATHER OF JANE—equivalent to the Prolog dyadic predicate FATHER(JOHN, JANE)
- ?80 WEATHER IS FINE—indicates an 80 percent plausibility that the assertion is true
- * = *—an axiom that contains a variable and defines equality to AP998

Rules with premises allow the system to deduce new facts from the facts defined to it. Examples of rules with premises are:

```
*X IS SON OF *Y IF *X IS MALE AND *Y IS
   PARENT OF *X
*X IS PARENT OF *Y IF *X IS FATHER OF *Y OR
   *X IS MOTHER OF *Y
?70 I WILL GO TO THE THEATER IF WEATHER IS
   FINE
```

As the examples show, rules are accepted by the system in a way very similar to natural language. The last example can be read in the following way: "There is a 70 percent plausibility that I will go to the theater if the weather is fine."

When the conclusion of a rule depends on uncertain premises, the following are applied:

1. The plausibility of two premises separated by AND is the minimum of the plausibilities of the individual premises.
2. The plausibility of the conclusion of the rule is the product of the plausibility of the rule times the plausibility of the premises, divided by 100.
3. If the plausibility of the conclusion is smaller than a certain threshold value, and the subgoal answered by the conclusion included a variable, this solution is abandoned (i.e., its plausibility becomes zero).
4. If two premises separated by OR carry to the same conclusion, both results are passed to APL separately (as independent answers to the same question).

Structure of the knowledge base. AP998 maintains information in two different data spaces. The first one is a symbol table, where words are stored. The other is the rule table. The size of each is automatically chosen by AP998 to fit all of the words and rules defined to it. Their starting (minimum) size is 2K bytes. Their maximum size is 63K bytes.

A stack is also used for logic inferences, the size of which can be adjusted by the programmer within the same interval. (The default size is 2K bytes.) Therefore, the total data space for AP998 may vary between 6K bytes and about 190K bytes. The information in the stack allows AP998 to provide information on *why* it came to a given conclusion.

The maximum number of rules accepted by AP998 is about 3000. Of course, this number depends on the rules themselves, for rules are variable-length objects, depending on the number and sizes of their premises.

Example. As an example of the use of AP998, we will solve the following logic problem, taken from Reference 12:

"When Alice entered the forest of forgetfulness, she did not forget everything, only certain things. She often forgot her name, and the most likely thing for her

Figure 1 AP998 solution to logic problem

```
/* Solution to the ALICE problem in AP998 */
/* Definition of YESTERDAY                */
sunday    is yesterday of monday
monday    is yesterday of tuesday
tuesday   is yesterday of wednesday
wednesday is yesterday of thursday
thursday  is yesterday of friday
friday    is yesterday of saturday
saturday  is yesterday of sunday
/* Data about the lion and the unicorn    */
The lion    lies on monday
The lion    lies on tuesday
The lion    lies on wednesday
The unicorn lies on thursday
The unicorn lies on friday
The unicorn lies on saturday
/* Data about the phrases they said       */
The lion can say that on * if
    the lion lies on *
    and *Y is yesterday of *
    and the lion lies not on *Y
The lion can say that on * if
    the lion lies on *Y
    and *Y is yesterday of *
    and the lion lies not on *
The unicorn can say that on * if
    the unicorn lies on *
    and *Y is yesterday of *
    and the unicorn lies not on *Y
The unicorn can say that on * if
    the unicorn lies on *Y
    and *Y is yesterday of *
    and the unicorn lies not on *
/* Finally, both the lion and the unicorn */
/* have said that today, so that          */
Today is * if
    the lion can say that on *
    and the unicorn can say that on *
```

to forget was the day of the week. Now, the lion and the unicorn were frequent visitors to this forest. These two are strange creatures. The lion lies on Mondays, Tuesdays, and Wednesdays, and tells the truth on the other days of the week. The unicorn, on the other hand, lies on Thursdays, Fridays, and Saturdays, but tells the truth on the other days of the week.

"One day Alice met the lion and the unicorn resting under a tree. They made the following statements:

```
LION:    Yesterday was one of my lying days
UNICORN: Yesterday was one of my lying days
```

"From these statements, Alice, who was a bright girl, was able to deduce the day of the week. What was it?"

The solution is given by the AP998 program in Figure 1.

The APL2/PC product also includes a workspace containing a set of cover functions that can be used with the AP998 auxiliary processor. Figure 2 is a sample of their use in solving the Alice problem.

Performance. The performance of the auxiliary processor when compared against the use of pure APL functions depends on the application, on the number of rules, and on the average search depth to solve a question. In the case of the Alice example just detailed, the average time to solve the problem is 18.5 milliseconds on a Personal System/2* with a 25 Mhz processor speed. The APL2 function in Figure 3 needed 38 milliseconds to get the same result.

Of course, in this simple case, where the loop can be eliminated completely, the difference is not very large. In a real case, with many more rules and a true cascade of results, the use of the auxiliary processor would provide a real performance improvement.

Figure 2 Cover functions used to solve logic problem

```
      ASK 'TODAY IS *'
THURSDAY
      WHY
I HAVE USED RULE NUMBER 18:
  TODAY IS THURSDAY IF
       THE LION CAN SAY THAT ON THURSDAY
       AND THE UNICORN CAN SAY THAT ON THURSDAY
I HAVE USED RULE NUMBER 15:
  THE LION CAN SAY THAT ON THURSDAY IF
       THE LION LIES ON WEDNESDAY
       AND WEDNESDAY IS YESTERDAY OF THURSDAY
       AND NOT THE LION LIES ON THURSDAY
I HAVE USED RULE NUMBER 10:
  THE LION LIES ON WEDNESDAY
I HAVE USED RULE NUMBER 4:
  WEDNESDAY IS YESTERDAY OF THURSDAY
I HAVE USED RULE NUMBER 16:
  THE UNICORN CAN SAY THAT ON THURSDAY IF
       THE UNICORN LIES ON THURSDAY
       AND WEDNESDAY IS YESTERDAY OF THURSDAY
       AND NOT THE UNICORN LIES ON WEDNESDAY
I HAVE USED RULE NUMBER 11:
  THE UNICORN LIES ON THURSDAY
I HAVE USED RULE NUMBER 4:
  WEDNESDAY IS YESTERDAY OF THURSDAY
```

Figure 3 APL2 function

```
[0]  Z←ALICE;LL;UL;LC;UC;DAYS;YEST
[1]   DAYS←'SUND' 'MOND' 'TUES' 'WEDN' 'THUR' 'FRID' 'SATUR'
[2]   YEST←¯1φDAYS
[3]   (LL UL)←('MOND' 'TUES' 'WEDN')('THUR' 'FRID' 'SATUR')
[4]   LC←((~DAYS∈LL)∧(YEST∈LL)) ∨ ((DAYS∈LL)∧(~DAYS∈LL))
[5]   UC←((~DAYS∈UL)∧(YEST∈UL)) ∨ ((DAYS∈UL)∧(~DAYS∈UL))
[6]   Z←(LC∧UC)/DAYS
```

APL and neural networks

A neural network (also called a "connectionist system") is a set of elementary units, called neurons, mutually related by means of connections. Each neuron has a certain number of inputs and a single output, which can divide itself to provide connections (inputs) to many other neurons. In addition, a certain real number is associated with each neuron (its threshold) and with each connection (its weight).

The response of a neuron is a procedure that computes the output of the neuron as a function of its inputs, the weights of its input connections, and the threshold of the neuron. Usually, the response of a neuron can be expressed in the following way:

$$f((\Sigma w_i x_i) - \Theta) \qquad (1)$$

where x_i is the set of inputs to the neuron, w_i represents the respective weights of the input connections, Θ is the neuron threshold, and f is the response function.

If the response function f can only have the values zero or one, the neuron is called digital. Otherwise, it is called analogic.

In typical neural networks, all the neurons have the same response function, and connections are such that the neurons can be divided into a certain number of layers. Neurons in the first layer (the input layer) have inputs that do not come from other neurons, but that come from outside the neural network (from the environment). Neurons in the last layer (the output layer) have outputs that do not go to other neurons, but go instead to the environment. There may be zero to any number of intermediate layers (also called "hidden layers").

A neural network where at least one neuron sends a connection to another neuron in a preceding layer is a neural network with feedback. An interesting family of neuron networks with feedback is called "Hopfield neural networks."[13]

A neural network with just two layers (one input layer and one output layer) and no feedback between them is called a *perceptron*. In an important paper, Minsky and Papert proved that it is impossible to generate the "exclusive-OR" operation with a perceptron.[14] Their paper effectively put an end to all research in neural networks for several years. Current research usually uses neural networks with one intermediate layer.

Matrix representation of a neural network. In general, any neuron in a neural network can provide an input (a connection) to any other neuron. Therefore, the network structure can be represented by a square n-by-n matrix, where n is the number of neurons in the network and the element i,j in the matrix is the weight of the connection from neuron i to neuron j. Nonexistent connections can be represented as connections of zero weight (since Equation 1 is not affected by those null connections).

The connection matrix represents the structure of the network. To include all of the available information we need an additional vector with the thresholds of all of the neurons in the network, given, of course, in the same order as in the matrix rows and columns.

However, if the response of all of the neurons in a network is of the form indicated by Equation 1, the network will be equivalent to another network. In that other network, all of the neurons in the original network are present, with the same connections and

weights, but with zero threshold, and an additional input neuron, whose output is always one, has been added. The additional input neuron is connected to every neuron in the network by means of a connection whose weight is equal to minus the threshold of the target neuron in the original network. The proof of this assertion is obvious from Equation 1.

Thus, a neural network with n neurons and arbitrary thresholds can be considered equivalent to another neural network with $n + 1$ neurons, all of them with zero threshold. Therefore, the behavior of any neural network can be represented by a single matrix if the response of its neurons corresponds to Equation 1.

We will represent the inputs as a vector of values which we will extend to the same length as the number of neurons in the network. This extension is easy. It is enough to assume that all of the neurons have exactly one input, and assign zero as the input value of those neurons that in actual fact did not have any input.

The output of the network can be computed by means of the following simple APL2 function:

```
[0]  Z←CONEC COMPUTE1 INPUT;A
[1]   Z←INPUT
[2]  L:A←Z
[3]   Z←(INPUT+A+.×CONEC)>0
[4]   →(~A≡Z)/L
```

The left argument is the connectivity matrix that defines the network. The right argument is the input vector. Note that the response function, applied to the whole neural network, is digital, and reduces in this case to an inner product and a comparison.

The preceding function has a loop because each inner product propagates the effect of the input to the next accessible layer. The loop, which proceeds until the network stabilizes, will eliminate the transient stages and provide us with the steady-state result. In a neural network without feedback, the loop will be executed at most n times, where n is the number of layers in the network, usually equal to three.

Analogic neurons. The neurons described in the previous subsection were digital, since their output can only be zero or one. Analogic neurons can produce other outputs, such as any number in the [0, 1] interval. For example, a commonly used response function for neural networks is:

$$1/(1+e^{-\Sigma w_i x_i}) \qquad (2)$$

with appropriate corrections when the value obtained is too near one or zero. The following APL2 function computes the result of a neural network composed of neurons with this response function. The neural network is assumed to be represented by a single connectivity matrix.

```
[0]  Z←CONEC COMPUTE2 INPUT;A
[1]   Z←INPUT
[2]  L:A←Z
[3]   Z←÷1+*-INPUT+A+.×CONEC
[4]   Z[(Z<0.2)/⍳⍴Z]←0
[5]   Z[(Z>0.8)/⍳⍴Z]←1
[6]   →(~A≡Z)/L
```

Learning. We say that a neural network "learns" when it modifies its behavior in such a way that its response to a certain set of inputs adapts to another set of predefined "desired outputs."

Different learning procedures modify the weights of the connections of the neural network in such a way that the outputs get closer and closer to the desired values. These techniques require a teaching period during which the following steps happen:

1. One or several inputs are applied to the network.
2. The corresponding outputs are computed.
3. The outputs are compared to the desired outputs.
4. The weights of the connections are modified so that the outputs become more like the desired outputs.

The above process is repeated until the network behavior is acceptable.

One of the learning procedures most used in neural networks is called "back propagation" because the weight corrections are applied to those output neurons in the last layer that differ from the desired value, and then the correction is propagated to the preceding layers. The APL2 program in Figure 4 executes a version of back propagation.

This program makes use of several global variables: CONEC is the matrix defining the neural network. LAYERS is a vector that contains the number of

Figure 4 Back propagation program

```
[0]   BKPROP1 I;INPUT;OUTPUT;O;OUT;E;d;NT;NO;ER;N;E1
[1]    E←0.02
[2]    NT←1++/N←LAYERS
[3]   L:'Input value: ',⍕IN[I;]
[4]    INPUT←1,IN[I;],(N[2]+N[3])⍴0
[5]    'Output value: ',⍕OU[I;]
[6]    OUTPUT←OU[I;]
[7]   L1:O←(-N[3])↑OUT←CONEC COMPUTE INPUT
[8]    ER←0.5×+/(E1←O-OUTPUT)*2
[9]    →(ER<1E¯10)/0
[10]   d←E×OUT∘.×E1
[11]   NO←1+N[1]+N[2]+⍳N[3]
[12]   d←d×CONEC[;NO]≠0
[13]   CONEC[;NO]←CONEC[;NO]-d
[14]   E1←(-NT)↑E1
[15]   NO←(∨/d≠0)/⍳↑⍴d
[16]   d←E×OUT∘.×(CONEC+.×E1)[NO]
[17]   d←d×CONEC[;NO]≠0
[18]   CONEC[;NO]←CONEC[;NO]-d
[19]   →L1
```

neurons in each layer. IN is a matrix of possible inputs. Finally, OU is the set of desired output values.

The program assumes that the number of layers in the network is three (the usual number). A few modifications would have to be done to apply a similar procedure to a perceptron or to a network with four or more layers.

Performance. In evaluating the performance of neural networks, there are two different considerations.

Performance of the learning process is one item. From the analysis of the back-propagation algorithm, it will be seen that the function contains an unavoidable loop. Therefore, the use of an interpreter (such as APL) will introduce a certain degradation. However, it must be remembered that the learning process is usually executed only once. After the neural network has learned successfully, it can be used many times without any further execution of the back-propagation algorithm or whatever else has been used. This means that the bottleneck is not so important unless the number of neurons is very large, and then APL may also have problems due to lack of space. But even this space problem can be solved, as the network connectivity matrices contain many zeros, and an implementation of sparse matrices can be used to make them fit in a given workspace.

Once the neural network has been trained, it will be applied to special cases, and this means that only the COMPUTE functions will be needed. It is easy to see that these functions also have a loop, but of a very different kind, since the number of times it is executed is equal to the number of layers in the network, which is usually equal to three. Therefore, interpretation time is negligible in this case as compared to the execution time of the inner product, where APL has no disadvantage as compared to a compilative program, since the inner product algorithm is a precompiled section of the interpreter.

APL and hypertext

The classical way of obtaining and presenting information is linear. In a book, or a written paper, or on the screen of a computer, the information is displayed as a succession of pages, each consisting of a number of lines, each line made of a succession of words. The reader will usually reach the desired information in a sequential process, by reading a word at a time, line by line, and page by page.

However, the use of certain "fast-reading" techniques allows the reader to browse the information in an extended way, overreaching the limits of the linear presentation. In an extreme case, rarely attained, we can consider that an ideally fast reader would be able to look at a page of text as a single unit, scanning it in a block and thus gaining a two-dimensional access to the information it contains.

What is hypertext? The term *hypertext*[15] has been applied to a recent means of information presentation that tries to transcend the limitations of the purely sequential display, allowing the reader a greater freedom in using scanning and retrieval procedures. The term was first applied in 1965 by Ted Nelson, who defined it as a hypothetical nonsequential writing tool.

All kinds of information can be combined to make up a hypertext application.

We can define hypertext as a nonlinear form of information presentation, where the units of information are the members of a hierarchy, linked in a certain way that makes it possible to attain very fast information retrieval. The search for an appropriate piece of data follows a nonlinear sequence directed by the train of thought of a reader, who is able to perform an associative navigation throughout the mass of information within reach. In this way, since it transcends the limitations of the written page, it can be said that hypertext provides the reader with a three-dimensional access to information.

The units of information in a hypertext system are usually the nodes of a hierarchical organization. The links that make up the hierarchy, which should be independent of the physical sequence of nodes, may be implicitly or explicitly defined by means of preprogrammed tags.

The benefits of hypertext are obvious. Besides the greater freedom provided to the reader by its three-dimensional access to information and its user friendliness, it is also quite easy to develop.

Hypertext media. All kinds of information can be combined to make up a hypertext application. We find:

- Visual information. This form is the most frequently used type in current computers. It consists of text, graphics, images, animation, video recordings, etc.
- Auditory information. This type includes speech and audio recordings.
- Other sensory data. At present, olfactory and tactile data are not usually found in computer applications, but perhaps in the future they will also be integrated into hypertext systems.
- Computer programs.

All of these kinds of information are kept in the ordinary physical storage media, such as fixed disks, diskettes, tapes, and compact discs.

Applications of hypertext. Hypertext methods can be applied wherever there is a need to manage large masses of information that can be divided into many chunks and accessed in a random way. For example:

- On-line documentation (help systems, reference works)
- Publishing (on-line dictionaries, computer-based encyclopedias)
- Computer-aided instruction (training manuals, tutorials, user guides)
- Expert systems, which require a highly developed interface to make use of the system so that it is friendly to a professional user who is not oriented to computer science (a physician, a lawyer, etc.)

Object-oriented programming and hypertext. Object-oriented programming (OOP)[16–18] is a programming method that is almost the exact opposite of classical procedural programming. In OOP, it is the *data* that are organized in a basic control hierarchy. One piece of data may be linked to another through a relation of descendancy, and this fact gives rise to a network (usually a tree) similar to the hierarchy of programs in procedural programming. There are also programs done in OOP, but they are appendages to the data (in the same way as in classical programming in which data are appendages of programs). It is possible to build global programs (accessible to all of the data in the hierarchy) and local programs (accessible from certain objects and their descendants).

In OOP, the execution of a program is fired by means of a *message* that somebody (the user, another program, or an object) sends to a given object. The recipient of the message decides which program should be executed. (It may be a local program or a global program which must be located through the network that defines the structure of the objects.)

Object-oriented programming is the appropriate way to program a hypertext application. In fact, the hierarchical data structure of OOP is the exact counterpart of the hierarchy of information units (the nodes) in hypertext. Hypertext links become the relations between objects in OOP. Hierarchical relations correspond to the links defining the hierarchy. Semantical relations provide the possibility of implementing other links that transcend the hierarchy.

The most generally used way to represent objects in object-oriented programming systems is by means of frames, a powerful data structure proposed by Minsky in 1975.[19] A frame system is a graph in which the nodes (frames) have a name and contain all of the information available about a given object. For example:

```
Frame TABLE
   Is_a: FURNITURE
   Files: 0,1,2
   Drawers: 0,1
   Legs: 4
   Light: 0,1
```

Object-oriented programming and APL2. In APL2, the existence of the general array makes it very easy to define and implement frames, which can be considered as general matrices of two columns, where the first element in each row contains a name and the second a (possibly multiple) value. For example, the frame mentioned above is a general matrix of five rows and two columns; it can be represented in APL2 in the following way:

```
TABLE ← 5 2 ⍴
   'IS_A' 'FURNITURE'
   'FILES' (0 1 2)
   'DRAWERS' (0 1)
   'LEGS' 4
   'LIGHT' (0 1)
```

With the use of frames, it is quite easy to build an object-oriented programming paradigm in APL2. Each object is represented as a frame, linked to other objects to form a hierarchy. The root of the hierarchy is called OBJECT and is initially defined as follows:

```
OBJECT ← 8 2 ⍴
   'PARENT' ''
   'CREATE'      'METHOD'
   'ERASE'       'METHOD'
   'PARENTS'     'METHOD'
   'CHILDREN'    'METHOD'
   'PROPERTIES'  'METHOD'
   'VALUE'       'METHOD'
   'METHODS'     'METHOD'
```

Each object in the hierarchy automatically inherits the properties and the methods defined by its ancestors (its parent and the ancestors of its parent), unless some property or method has been redefined, either by the same object or by a lower-level ancestor. The inheritance of methods and the ability to send messages to any object are easily implemented by means of the APL2 function MESSAGE, with the syntax:

```
MESSAGE 'Object' 'Method'
   [additional information]
```

and the implementation shown in Figure 5.

References 20 and 21 explain in more detail the applicability of APL2 for object-oriented programming. Thus, we can deduce that object-oriented programming in APL2 is a good way to program a hypertext application.

An on-line dictionary written in APL2 (OOP). A part of a Spanish on-line dictionary for the high-school level has been implemented in APL2/PC using object-oriented programming techniques. The dictionary currently contains the definitions of 2130 words in science and technology, distributed in the following fields:

- Biographies (123)
- Computer science (18)
- Technology (338)
  - Electronics (71)
  - Materials (59)
  - Vehicles (54)
  - Instruments (78)
  - Miscellaneous (76)
- Medicine (409)
- Biology (1100)
  - Anatomy (244)
  - Physiology (120)
  - Cytology and histology (38)

Figure 5 Implementation of MESSAGE

```
[0]   ΔR←MESSAGE ΔX;ΔOB;ΔMET;ΔSRCH;ΔA;ΔI;ΔB
[1]   ΔOB←' ' ELM↑ΔX
[2]   ΔMET←' ' ELM 2⊃ΔX
[3]   ΔX←2↓ΔX
[4]   →ΔE1 IF~EXIST ΔOB
[5]   ΔSRCH←ΔOB
[6]  ΔL:ΔA←(ΔB←GET ΔSRCH)[;1]
[7]   →ΔL1 IF(⍴ΔA)≥ΔI←ΔA⍳⊂ΔMET
[8]   →ΔE2 IF 0=⍴ΔSRCH←⊃ΔB[ΔA⍳⊂'PARENT';2]
[9]   →ΔL
[10] ΔL1:ΔX←(⊂ΔOB),ΔX
[11]  '' ⎕EA 'ΔR←',ΔSRCH,'_',ΔMET,' ΔX'
[12]  →0
[13] ΔE1:ΔMSG 'THE OBJECT' ΔOB 'DOES NOT EXIST. METHOD =' ΔMET
[14]  →0
[15] ΔE2:ΔMSG 'UNKNOWN METHOD' ΔMET 'FOR OBJECT' ΔOB
```

- Genetics (14)
- Biochemistry (78)
- Ecology (23)
- Paleontology (40)
- Microbiology (28)
- Zoology (incomplete) (302)
- Botany (incomplete) (140)
- Miscellaneous (73)

• Others (142)

The OOP application consists of a total of 2133 objects, three of which (the root of the hierarchy) are in the APL2/PC workspace, whereas the others (the words in the dictionary) are included in 44 files, created and used by means of the AP211 auxiliary processor.[22,23] The total size of these files is 1 372 216 bytes, which makes an average of 644 bytes per word definition, 31 187 bytes and 48 words per file. Words are distributed in the files thematically to reduce the overhead, since it can be assumed that groups of words searched in the dictionary will usually be related in this way. Therefore, not all files are equal in size, the largest one consisting of 142 words and 93K bytes, and the smallest one consisting of 8 words and 4K bytes.

Summary

This paper and others in the references show the usefulness of APL and APL2 for the most modern programming techniques and applications. Among these applications are artificial intelligence, neural networks, object-oriented programming, and hypertext, which have been described in some detail.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. J. A. Brown and M. Alfonseca, "Solution to Logic Problems in APL2," *SEAS Spring Meeting 1987*, Vol. 2 (1987), pp. 819–829.
2. *APL2 Programming: Language Reference*, SH20-9227, IBM Corporation (1987); available through IBM branch offices.
3. J. A. Brown, "APL2—A New Comer on the Artificial Intelligence Scene," *SEAS Spring Meeting 1986*, Vol. 2 (1986), pp. 819–830.
4. R. Guerreiro, "APL2 and LISP," *SEAS Anniversary Meeting 1988*, Vol. 2 (1988), pp. 1435–1460.
5. J. Ansell and M. Al-Doori, "Towards an Expert System," *APL-ication Proceedings* (1989), pp. 65–72.
6. D. Smellie and F. Evans, "A Structured Approach to Building Expert Systems," *APL-ication Proceedings* (1989), pp. 86–99.
7. A. Prys-Williams, "Expert Systems with Existing Software," *VECTOR* **5**, No. 3, 57–74 (January 1989).
8. P. Rodríguez, J. Rojas, and M. Alfonseca, "An Expert System in Chemical Synthesis Written in APL2/PC," *APL89 Conference Proceedings*, *APL Quote Quad* **19**, No. 4, 293–298 ACM, New York (1989).
9. K. Fordyce, J. Jantzen, G. A. Sullivan, Sr., and G. A. Sullivan, Jr., "Representing Knowledge with Functions and Boolean Arrays," *IBM Journal of Research and Development* **33**, No. 6, 627–646 (November 1989).
10. J. A. Brown, E. Eusebi, L. Groner, and J. Cook, *Algorithms and Artificial Intelligence in APL2*, Technical Report TR 03.281, IBM Corporation, Santa Teresa Laboratory, San Jose, CA (1986).

11. M. J. Tobar and M. Alfonseca, "Emulating Prolog in a PC APL Environment," *APL86 Conference Proceedings, APL Quote Quad* **16**, No. 3, 13–15, ACM, New York (1986).
12. R. Smullyan, *What Is the Name of This Book?*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1978).
13. J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proceedings of the National Academy of Science* **79**, 2554–2558 (1982).
14. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA (1965).
15. *Hypertext: Theory into Practice*, Ray McAleese, Editor, Blackwell Scientific Publications, Oxford (1989).
16. B. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Co., Reading, MA (1986).
17. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).
18. *Object-Oriented Computing*, G. Peterson, Editor, Vol. 1 & 2, IEEE Order No. 821 and 822 (1987).
19. M. Minsky, "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, P. Winston, Editor, McGraw-Hill Book Co., Inc., New York (1975), pp. 211–217.
20. M. Alfonseca, "Object Oriented Programming in APL2," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 3, 6–11, ACM, New York (1989).
21. M. Alfonseca, "Frames, Semantic Networks, and Object-Oriented Programming in APL2," *IBM Journal of Research and Development* **33**, No. 5, 502–510 (September 1989).
22. *APL2 for the IBM Personal Computer*, Program Number 5799-PGG, PRPQ RJB411, Part No. 6242036, IBM Corporation; available through IBM branch offices.
23. M. Alfonseca and D. Selby, "APL2 and PS/2: The Language, the Systems, the Peripherals," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 4, 1–5, ACM, New York (1989).

Accepted for publication June 10, 1991.

Manuel Alfonseca *IBM Software Technology Laboratory, Paseo de la Castellana, 4, 28046 Madrid, Spain.* Dr. Alfonseca is a Senior Technical Staff Member in the IBM Software Technology Laboratory. He has worked in IBM since 1972, having been previously a member of the IBM Madrid Scientific Center. He has participated in a number of projects related to the development of APL interpreters, continuous simulation, artificial intelligence, and object-oriented programming. Eleven international IBM products have been announced as a result of his work. Dr. Alfonseca received electronics engineering and Ph.D. degrees from Madrid Polytechnical University in 1970 and 1971, and the Computer Science Licenciature in 1972. He is a professor in the Faculty of Computer Science in Madrid. He is the author of several books and was given the National Graduation Award in 1971 and two IBM Outstanding Technical Achievement Awards in 1983 and 1985. He has also been awarded as a writer of children's and juvenile literature.

Reprint Order No. G321-5453.

Language as an intellectual tool: From hieroglyphics to APL

by D. B. McIntyre

We learn elementary mathematics before understanding the source of its symbols and procedures, which therefore appear, incorrectly, to have been decreed ready-made. Language and reason are intimately related, and the embodiment of an idea in a symbol may be essential to its comprehension. APL unifies algebra into a single consistent notation; it allows us to exploit the powerful concepts of functions and operators; and it helps us to escape from the tyranny of scalars by giving us the tools to think in terms of arrays, or multiple quantity, as J. J. Sylvester so eloquently urged us to do a century ago. APL has an intellectual consistency that is a source of satisfaction and pleasure. This paper traces the history of symbols from hieroglyphics to APL.

The APL language, a language with symbols and not words, is one of the intellectual triumphs of our time. Its modern incarnation began with Iverson notation,[1,2] but its roots go far back into the past.

In the beginning

Perhaps the earliest record of what came to be APL was carved on a sculptured mace of granite about 3100 BC, before the invention of papyrus. Of course you cannot read it, unless as is the case with contemporary APL, you know the meaning of the symbols.

We shroud in mystery whatever we do not understand. In crystal optics we speak of "extra-ordinary" rays, though there is, of course, nothing extra-ordinary about them. Negative numbers were called absurd or fictitious. Even after Leonardo of Pisa (known as Fibonacci), in the year 1202, had taught us to recognize debt as a negative asset, it took another 400 years before the number scale was represented geometrically. Intellectual progress is slow, and an additional 250 years passed before Sylvester showed how absurd it was to style as imaginary the quantities represented by the symbols i, j, k of "complex" numbers and quaternions.

I remind you of the words of Whitehead: "Mathematics is often considered to be a difficult and mysterious science, because of the numerous symbols which it employs. Of course, nothing is more incomprehensible than symbolism we do not understand."[3]

The inscription illustrated in Figure 1 is a record of the triumph of Menes, founder of the first dynasty

Figure 1 An example of early hieroglyphics

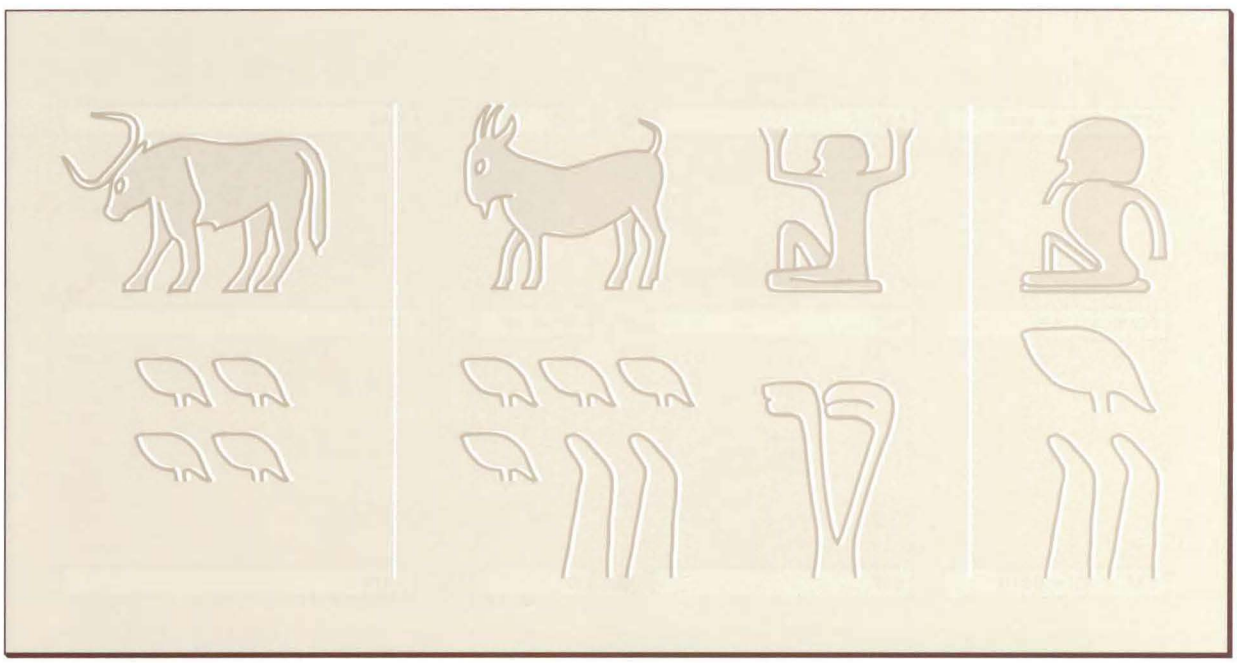
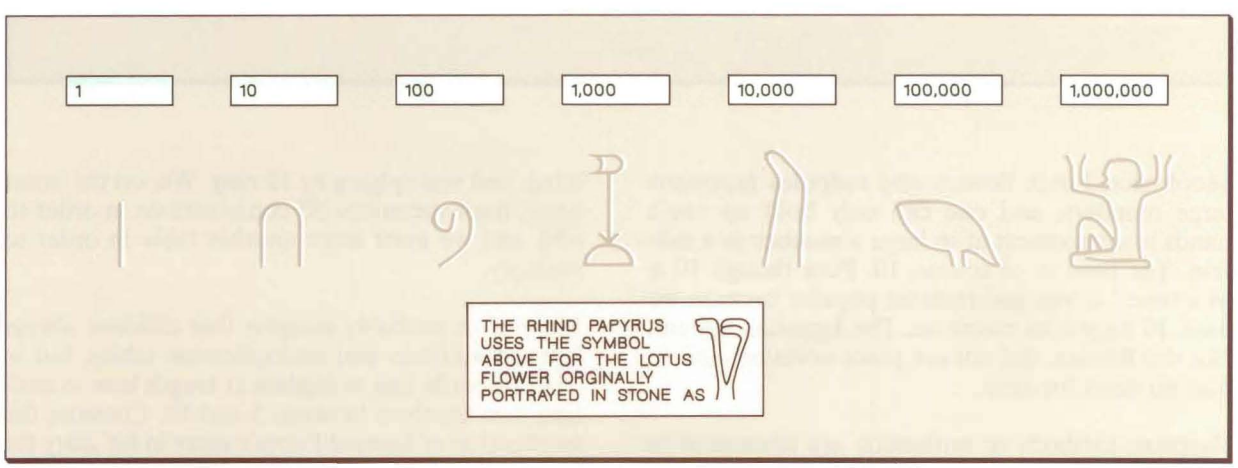


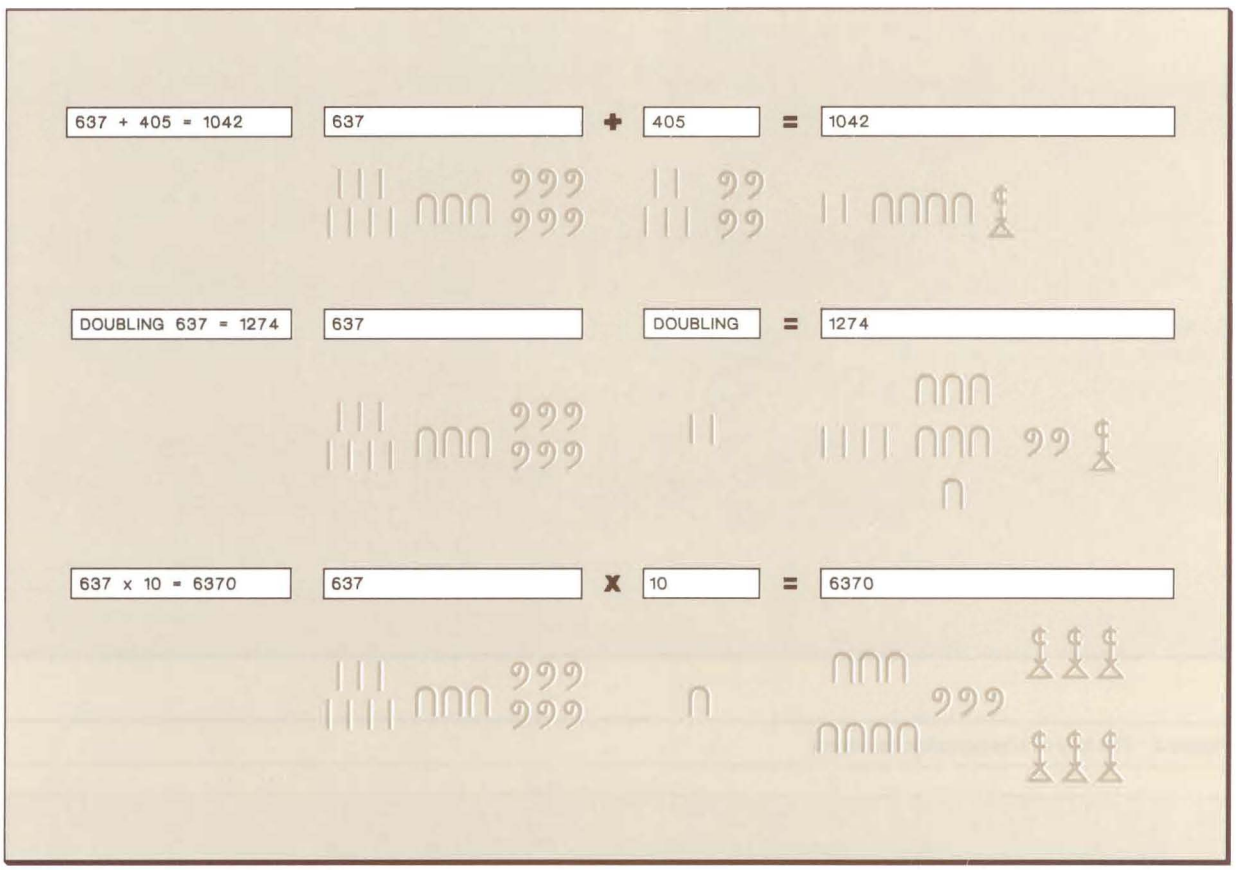
Figure 2 The key to hieroglyphic numbers



of historical pharaohs, who united the two kingdoms of Egypt. With Figure 2 as our key, we read that he captured 400 000 oxen, 1 422 000 goats, and 120 000 prisoners.

Although the variables are named, the example lacks the equivalent of APL's assignment arrow. A hundred is represented in hieroglyphics by a picture of the coiled rope used by Egyptian surveyors, or "rope-stretchers," whose descendants today use the "chain" as a unit of measurement. We should remember that Eratosthenes, the director of the great library in Alexandria, was the first to measure the earth's circumference, thus initiating the science of

Figure 3 Examples of Egyptian methods of arithmetic



geophysics. Lotus flowers and tadpoles represent large numbers, and one can only hold up one's hands in amazement at so large a number as a million. The base is, of course, 10. Poor though 10 is as a base,[4] it was and remains popular because we have 10 fingers to count on. The Egyptian system, like the Roman, did not use place notation, and so had no need for zero.

Egyptian methods of arithmetic are illustrated in Figure 3, reading the symbols from right to left, i.e., the more significant figures are to the right. The three examples represent: adding 637 and 405; doubling 637; and multiplying 637 by 10. The system has been derided as clumsy, but for more than a thousand years no nation was able to improve on the Egyptian notation and methods.[5] Again, Figure 2 is the key to understanding the notation in Figure 3. This system makes addition, subtraction, doubling, and multiplying by 10 easy. We, on the other hand, must memorize 55 combinations in order to add, and we must learn another table in order to multiply.

Most of us probably imagine that children always learned addition and multiplication tables, but in 1542 Recorde had to explain at length how to multiply two numbers between 5 and 10. Consider the implication of Samuel Pepys's entry in his diary for July 4, 1662: "Comes Mr. Cooper of whom I intend to learn mathematiques, and do begin with him today. After an hour's being with him at arithmetique, my first attempt being to learn the multiplication table." Five days later he records: "Up by four o'clock, and at my multiplicacion-table [sic] hard, which is all the trouble I meet withal in my arithmetique." Now Pepys was a 30-year-old graduate of Cambridge, an able man of business, soon

Figure 4 First appearance of symbols in print

Plus, Minus	$+ \; -$	Johann Widman	1489
Equals	$=$	Robert Recorde	1557
Times	$\times$	William Oughtred	1631
Greater than, Less than	$> \; <$	Thomas Harriot	1631
Exponentiation	A^{iii}	James Hume	1636
Greater than or equal to, Less than or equal to	$\geq \; \leq$	John Wallis	1655
Divide	$\div$	Johann H. Rahn	1659
Summation	Σ	Leonard Euler	1755
Factorial	$!$	Christian Kramp	1808
Absolute Value	$\lvert A \rvert$	K. Weierstrass	1841
Membership	$\in$	Giuseppe Peano	1889
Logical Not	$\sim$	Giuseppe Peano	1893
Inclusive Or	$\vee$	Whitehead and Russell	1909
And	$\wedge$	Alfred Tarski	1933

to become a Fellow of the Royal Society (as president of the society, Pepys gave his imprimatur to Newton's "Principia"). As Secretary of the Navy he became one of the nation's leading financiers.

How seldom do we look back in maturity at what we learned by rote as children, and that is why I like the title (as well as the content) of Klein's *Elementary Mathematics from an Advanced Standpoint.*[6] We are taught as if the common mathematical symbols came to humankind in antiquity engraved on stone; as if they had no history. The dates when some of these symbols first appeared in print show that our notation evolved over centuries[7,8] (see Figure 4). The imprints on our bank checks show that in our own time technology has changed some of our familiar symbols.

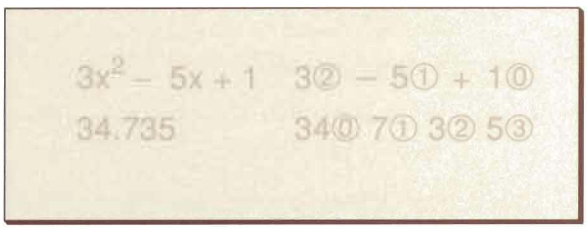
The acceptance of symbols

The symbol for *plus* is probably an abbreviation for the Latin *et*, and that for *minus* may be "a simple bar used by merchants to separate the indication of the tare, for a long time called 'minus,' from that of the total weight of the merchandise."[9] De Morgan thought the symbols might be marks on sacks or barrels showing whether they were over or under weight. Recorde, in 1557, first used these signs in an English book, the same one in which he gave us the *equals* symbol, which he chose "because noe 2 thynges can be moare equalle." Euler's Σ (sigma) suggests *summation*; epsilon is the first letter of the Greek *esti* (is a), which suggests *membership*; and the symbol for *or* is the first letter of the Latin *vel*.

In his survey of the development of mathematics, Kline pointed out that Leibniz "certainly appreciated the great saving of thought that good symbols make possible. Thus by the end of the seventeenth century, the deliberate use of symbolism—as opposed to incidental or accidental use—and the awareness of the power and generality it confers entered mathematics."[10]

Our notation having been at least 500 years in the making, it is no surprise that the story is not yet at an end. What is remarkable is that Iverson is apparently the first to look at the consistency and completeness of the notation as a whole. Function syntax is inconsistent; e.g., *summation* has its argument to the right, *factorial* to the left, and *absolute value* is written on both sides of its argument. *Exponentiation* has no symbol at all; its second argument is merely written as a superscript. Iverson also considered which other functions have sufficient utility to warrant separate graphic symbols. He showed that function names should not be elided, and pointed out the advantage of each symbol rep-

Figure 5 Exponents enclosed in circles (Stevinus, 1585)



resenting related monadic and dyadic functions. Iverson simplified syntax by abandoning function hierarchy (originally imposed for writing polynomials) and making each function take everything to its right as its right argument.

Acceptance of good symbols has, however, never been easy. After introducing the *times* symbol (Saint Andrew's cross) in 1631, Oughtred wrote: "This manner of setting downe Theoremes, whether they be Proportions, or Equations, by Symboles or notes of words, is most excellent, artificiall, and doctrinall [i.e., serving to teach]. Wherefore I earnestly exhort every one, that desireth though but to looke into these noble Sciences Mathematicall, to accustome themselves unto it: and indeede it is easie, being most agreeable to reason, yea even to sence. And out of this working may many singular consectaries [i.e., conclusions] be drawne: which without this would, it may be, for ever lye hid."[11]

But 15 years later, still more encouragement was needed: "[My] Treatise being not written in the usuall synthetical manner, nor with verbous expressions, but in the inventive way of Analitice, and with symboles or notes of things instead of words, seemed unto many very hard; though indeed it was but their owne diffidence, being scared by the newness of the delivery; and not any difficulty in the thing it selfe. For this specious [i.e., pleasing to the eye] and symbolicall manner, neither racketh the memory with multiplicity of words, nor chargeth the phantasie with comparing and laying things together; but plainly presenteth to the eye the whole course and processe of every operation and argumentation."[11]

It seems that not much has changed, judging from the experience of Giuseppe Peano (who provided two of APL's symbols). We are told that he "used a great deal of symbolism because he wished to sharpen the reasoning. . . . Peano used this symbolism in his presentation of all of mathematics, notably in his Formulario mathematico (5 vols., 1895–1908). He used it also in his lectures, and his students rebelled. He tried to satisfy them by passing all of them, but that did not work, and he was obliged to resign his professorship at the University of Turin."[12]

Smith, quoting Nesselmann's *Algebra of the Greeks* (1842), says that mathematics evolves through three stages: *rhetorical*, with words and sentences in full; *syncopated*, in which words are condensed by abbreviation; and *symbolic*, in which there are no words at all.[13,14] Consider the way we write equations. Comparison of 20 examples from 1463 to 1693[15] shows how long it took to pass from words to our present symbolic system. Simon Stevin (Stevinus, 1548–1620), for instance, made great progress by identifying exponents, writing them enclosed in circles (See Figure 5). His books (1585, 1586) were influential in promoting the use of the new methods. (See Reference 16.)

The superscript method of denoting a to the power b (that is, a^b) was used by Hume in 1636, though his use of Roman numerals for the exponent shows he thought only of integer powers. The form we use now was first used by Descartes in 1637. John Wallis, a distinguished predecessor of Sylvester's as Savilian Professor of Geometry in Oxford, was one of the first to write equations in the form we use today, though even he often wrote *aaaa* for a^4. Until the end of the eighteenth century it was, indeed, common practice to write *aa* for a^2. Wallis, who gave us our symbols for *greater-than-or-equal-to* ($\geq$) and *less-than-or-equal-to* ($\leq$) and our symbol for infinity (∞), found a meaning for negative exponents (1655, 1657), but Newton was the first to permit the exponent to be positive, negative, integer, or fractional (1676).

Euler, in 1777, introduced the symbol i (impossible or imaginary) for $\sqrt{-1}$, and by 1837 Sir William Rowan Hamilton had so adopted the geometrical interpretation of complex numbers (Wessel, Gauss, Argand) that it could be said that exponentiation had been extended to the case of a negative number with a fractional exponent. Cayley further extended the scope of exponentiation by raising matrices to positive integer powers and to the power $\neg 1$, which he called the "inverse or reciprocal" matrix.[17,18] Today's APL handles all these cases directly.

To indicate that a word was abbreviated, the practice used to be to put a stroke (solidus) through the last letter. This accounts for the lines still seen in

symbols for the British pound (£, Latin *libra*), the dollar ($, an abbreviation of *pesos*), the cent (¢), and the sign ℞ (for the Latin *recipe*, or the imperative "take") displayed by pharmacists.[19,20] Cardan used ℞ for "root" (Latin *radix*) in 1539, and we still talk of "extracting" (pulling out) the root. Although Euler believed the square root symbol (√) to be the deformed letter *r* (abbreviating *radix*), Cajori doubts this, suggesting its origin might be a dot.[21]

We are taught that it is a simple step from exponents to logarithms, and few developments have been more important. Laplace recognized our immense debt to Napier in his well-known remark about logarithms, that, by halving the labor, they had doubled the life of the astronomer and mathematician; but we seldom think of the primitive state of the conceptual tools available in 1614, or recognize Napier's genius. In his day, algebra differed little from arithmetic, and the notation we take for granted was almost nonexistent. Napier's discovery came three years before he invented the decimal point, and less than 60 years after Recorde introduced the equals sign and first used the signs + and − in an English book. Just how Napier succeeded in calculating his table of logarithms is well described by Gittleman.[22]

In a volume commemorating the 300th anniversary of Napier's *Description of the Marvellous Canon of Logarithms*, Glaisher well expressed the power of good notation: "Nothing in the history of mathematics is to me so surprising or impressive as the power it has gained by its notation or language. . . . By his invention [of logarithms] Napier introduced a new function into mathematics. . . . When mathematical notation has reached a point where the product of n xs was replaced by x^n, and the extension of the law $x^m \cdot x^n = x^{m+n}$ has suggested $x^{1/2} \cdot x^{1/2} = x$, so that $x^{1/2}$ could be taken to denote the square root of x, then the fractional exponents would follow as a matter of course, and the tabulation of x in the equation $10^x = y$ for integral values of y might naturally suggest itself as a means of performing multiplication by addition. But in Napier's time, when there was practically no notation, his discovery or invention was accomplished by mind alone without any aid from symbols."[23] (See also Reference 24.)

"We who live in an age when algebraical notation has been extensively developed can realise only by an effort how slow and difficult was any step in mathematics until its own language had begun to arise, and how great was the mental power shown in Napier's conception and its realisation. . . . In our days when the rules of computation are precise, and when the construction of instruments has reached a high state of efficiency, the processes of multiplication and other arithmetical operations can be performed by machines designed for the purpose. These apparatuses which save mental strain and time are effective aids to calculation, and they may be regarded as the modern successors to Napier's rods."[23]

APL and functional programming

APL's concise notation helps us grasp the intellectual content of an algorithm without the distraction of extraneous and irrelevant matters prescribed by a machine. APL is a succinct and admirably consistent language that not only uses verbs (functions) to act on nouns (data arrays), but uses adverbs and conjunctions (operators) to derive new verbs, and permits definition of new verbs, adverbs, and conjunctions. It has the subtlety and suggestiveness which, as Bertrand Russell said, makes a good notation "seem almost like a live teacher,"[25] and, to quote Pledge, "Suggestiveness is the essential service of symbolism."[26]

With APL, the goal of *functional programming* (Backus, 1978) can be achieved. The word *function* (derived from *functio*, meaning a performance or execution) was used at the end of the 17th century by mathematicians writing in Latin. Leibniz, who gave us many terms such as *constant*, *variable*, and *parameter*, used "function" in our sense in 1673. Euler used the symbol *f* for a function in 1734, and in 1754 used the notation *f:(a,n)* for a function of the variables *a* and *n*, i.e., to state that the result depends upon the current values of *a* and *n*. Iverson does better than this; in 1976 his method of *direct definition*[27] of functions shows formally exactly how the result is derived from the arguments, and Euler's parentheses are not needed.

The relationship between ordinary APL and direct definition is illustrated by the following examples:

In ordinary APL:

```
     ∇Z← A PLUS B
[1]    Z← A + B
[2]  ∇

      3 PLUS 4
7
```

```
     ∇Z← F N
[1]   ⍎(N=0)/'→0, 0ρZ←1'
[2]   Z←N× F N-1
[3]  ∇

      F 4
24
```

In direct definition:

```
      PLUS: α + ω

      3 PLUS 4
7
      F: ω × F ω-1 : ω=0 : 1
F
      F 4
24
```

The left and right arguments are denoted α and ω. The recursive definition of the factorial should be read: "The factorial of ω is ω times the factorial of $\omega-1$ unless ω equals zero, in which case the factorial of ω is 1."

To illustrate the advantage of Iverson's method, consider the problem of cluster analysis. Each entity, described by n variables, can be considered a point in n-dimensional space, and we are required to compute the distance between each point and all the others. If n is 2, the data are given in a matrix of two columns. We then represent each entity as a point, with coordinates x and y, plotting the points on a scatter diagram. The theorem of Pythagoras lets us determine the distance between any two points, and the results complete a square matrix. This *similarity matrix* gives the *closeness* of each entity to every other one based on all measured properties. The matrix is symmetric with zeros on the diagonal. In APL the algorithm automatically extends to higher dimensions.

Hellerman used this as an example of APL notation, in a book that (in both of its editions) is a landmark in the history of APL.[28] His solution is as follows:

```
     ∇Z←DISTANCE P;N;I;J
[1]   N←(ρP)[0]
[2]   D←(N,N)ρ0
[3]   J←0
[4]  L0:I←0
[5]  L1:D[I;J]←(P[I;]-P[J;])
       +.×(P[I;]-P[J;])
[6]   →(N>I←I+1)/L1
[7]   →(N>J←J+1)/L0
     ∇
```

Direct definition allows this to be expressed in a single line:

```
DSQ: (0 2 1 2⍉ω∘.-ω)+.*2
```

If this line seems strange or unduly terse to someone new to APL, I would point out that if we already know how to add, subtract, and square numbers, there are only three APL functions to learn: inner and outer products and dyadic transposition. I remember Adin Falkoff saying that good notation cannot make an inherently recondite concept easy, but it can remove unnecessary impediments by expressing the concept in as simple a manner as possible: Einstein's $E = m \times c^2$ is a simple statement of a relationship that probably can be fully understood by very few.

For a further illustration consider eight statistical functions, first in standard APL notation:

```
     ∇Z←MEAN X
[1]   Z←(+/X) ÷ 0⊥ρX
[2]  ∇

     ∇Z←DEV X
[1]   Z←X- (MEAN X)∘.+ (0⊥ρX)ρ0
[2]  ∇

     ∇Z←SS X
[1]   Z←(DEV X)+.*2
[2]  ∇

     ∇Z←VAR X
[1]   Z←(SS X)÷ ¯1+0⊥ρX
[2]  ∇

     ∇Z←SD X
[1]   Z←(VAR X)*0.5
[2]  ∇

     ∇Z←SP X;M
[1]   Z←M+.× ⍉M← DEV X
[2]  ∇

     ∇Z←COV X
[1]   Z←(SP X)÷ ¯1+0⊥ρX
[2]  ∇

     ∇Z←COR X;S
[1]   Z←(COV X)÷S∘.×S←SD X
[2]  ∇
```

They define the means, deviations from the means, sums of squares of the deviations, variances, standard deviations, sums of cross products, covariances, and correlation coefficients.

The functions form a pedagogic sequence in the sense that to understand any one of them you must first understand those that precede it. Each function can be directly defined in a single line, and each takes the original data as its argument.

Next, in direct definition:

```
MEAN:(+/ω)÷0⊥ρω
DEV:ω-(MEANω)∘.+(0⊥ρω)ρ0
SS:(DEVω)+.*2
VAR:(SSω)÷¯1+0⊥ρω
SD:(VARω)*0.5
SP:M+.×⍉M←DEVω
COV:(SPω)÷¯1+0⊥ρω
COR:(COVω)÷S∘.×S←SDω
```

Using Iverson's new dialect J,[29–31] the same functions can be defined even more succinctly, and without parentheses. Not only are no variables assigned, no explicit reference is made to the arguments. This is *tacit definition*, or pure functional programming (Backus, 1978), which leads to efficient execution and invites parallel processing. (Version 3.3 of Iverson's J is used for the examples that follow.)[32]

```
mean=.+/%#

dev=.-mean

ss=.+/@*:@dev

var=.ss%<:@#

sd=.%:@var

sp=.+/ .*~|:@dev

cov=.sp%<:@#

cor=.cov%*/~@sd
```

The sequence of functions starts with the mean and ends with the correlation coefficient. Is this structured programming? Is it top down or bottom up? Such questions seem to vanish in a sequence that is almost self-documenting.

The style of programming brings to mind the words of Babbage: "The almost mechanical nature of many of the operations of Algebra, which certainly contributes greatly to its power, has been strangely misunderstood by some who have even regarded it as a defect. When a difficulty is divided into a number of separate ones, each individual will in all probability be more easily solved than that from which they spring. In many cases several of these secondary ones are well known, and methods of overcoming them have already been contrived: it is not merely useless to re-consider each of these, but it would obviously distract the attention from those which are new: something very similar to this occurs in Geometry; every proposition that has been previously taught is considered as a known truth, and whenever it occurs in the course of an investigation, instead of repeating it, or even for a moment thinking on its demonstration, it is referred to as a known datum. It is this power of separating the difficulties of a question which gives peculiar force to analytical investigations, and by which the most complicated expressions are reduced to laws and comparative simplicity."[33]

Revisiting our roots

Being aware of the long history of functions in mathematics, and having seen examples written in current APL, we can now use APL to illuminate our roots, which reach back to Egyptian hieroglyphics. The word *algorithm*, according to the Oxford English Dictionary, is an erroneous refashioning of *algorism*, a word derived from "al-Khowarazmi, the native of Khowarazm, surname of the Arab mathematician who flourished early in the 9th Century, and through the translation of whose work on Algebra, the Arabic numbers became generally known in Europe." In its original form it was used by Chaucer, and the Oxford dictionary cites the use of *algorithm* in 1774.[34,35] I found it first used by Sylvester[36] in one of the earliest papers to speak of *matrices* (compare References 27 and 37 for APL treatment of polygons and polyhedra).

The earliest known book of algorithms is the Rhind Papyrus, based on work written 2000–1800 BC and copied by Ahmes the scribe in 1650 BC.[38–40] It is a textbook on solving practical problems. Consider a simple example, shown in Figure 6 and using Figure 2, again, as the key; to multiply 12 by 12 begin by writing down 12, and by successive doublings obtain 1, 2, 4, and 8 times 12. Check the rows 4× and 8× (on the papyrus the check marks are red) and add them to get the required result. The symbol preceding the answer is a rolled-up scroll (*quod erat demonstrandum*), which in fancy we may take as the ancestor of our *equals* and APL's *assignment* symbols.[41]

Figure 6 Rhind Papyrus problem 32; multiplying 12 by 12

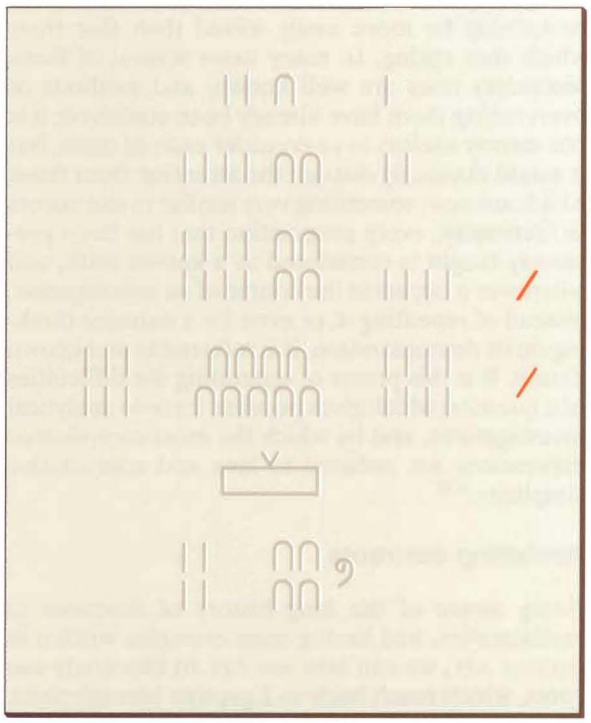
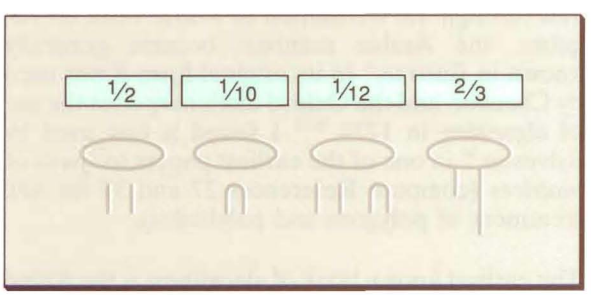


Figure 7 Hieroglyphic method of showing fractions



IBM's System/360* and its descendants use this ancient method to multiply integers. Microcode for fixed-point multiplication builds the 1×, 2×, 3×, and 6× products of the multiplicand in local storage. Then, just as the scribes did nearly 4000 years ago, it combines the products corresponding to the multiplier. If the multiplier is 8 or more, a shift of 4 is first made (corresponding to multiplication by 16), and then products are subtracted rather than added; e.g., to multiply by 11, first shift to multiply by 16, then subtract 6× and add 1×. One may ask why the products used by System/360 are 1×, 2×, 3×, and 6× instead of the 1×, 2×, 4×, 8× used by the Egyptians. When I raised this question in a lecture in New York in 1982, John Macpherson (who was the first to implement binary coded decimal on an IBM computer) gave me the explanation in engineering terms.

However unfamiliar its symbols may be to us, the hieroglyphic message is inherently simple. So it is with the symbols of APL, all of which stand for well-known or easily understood operations. Many today, as Oughtred found 350 years ago, are "scared by the newness of the delivery; and not by any difficulty in the thing itself"!

The ancient Egyptians used mathematics for practical purposes, such as paying wages and collecting taxes. Consider the instructive example of salary distribution at the Temple of Illahun—not paid in salt (as the word "salary" implies) but in jugs of beer and loaves of bread. Division, of course, often produces fractions, and the hieroglyphic way to represent fractions can be seen in Figure 7.

All fractions were represented as unit fractions, i.e., with a numerator of 1. Even 2/3, which seems like an exception, was represented as the unit fraction 1/1.5. The eye-like symbol is perhaps the earliest of all APL function symbols. It is the reciprocal, or *monadic divide*, which in APL has become an eye closed into a slit, with dots above and below (÷).

If a loaf of bread is divided into 10 parts, and you are to get 1 share, your portion is 1/10; if you are to get 2 shares your portion is 1/5; and if you are to get 5 shares your portion is 1/2. From these simple fractions, other shares can be computed by combination.[42] For example, 3 shares are the same as 1 + 2 shares, i.e., 1/5 + 1/10; 4 shares are the same as 2 + 2 shares, i.e., 1/5 + 1/5, which, by consulting a table of values of 2/n, is set down as 1/3 + 1/15.

Sylvester became interested in the unit fractions of the Egyptians when reading "the chapter in Cantor's *Geschichte der Mathematik* which gives an account of the singular method in use among the ancient Egyptians for working with fractions. It was their curious custom to resolve every fraction into a sum of simple fractions according to a certain traditional method, not leading, I need hardly say, except in a few of the simplest cases, to the expansion under the special form to which I have the name of a fractional sorites."[43]

Sylvester's algorithm is expressed in APL with tolerance as left argument:

```
     ∇Z←T F X
[1]    ⍎(X≤T)/'→0,0⍴Z←⍳0'
[2]    Z←Z, T F X-÷Z←⌈÷X
[3]  ∇
```

Sylvester's example is:

```
      1E¯16 F 335÷336
2 3 7 48
```

In direct definition, this leads to a useful paradigm for writing recursive functions in APL:

```
      F:Z,αFω-÷Z←⌈÷ω : ω≤α ⍳0

      (○1) = +/÷1E¯16 F ○1
1
```

Roger Hui (in a personal communication) translated this into the *purely functional* form in J, using @. for *agenda*:

```
  f=. i.@0: ' (>.@% @ ] , [f ]->.&.%@]) @.<:

  1e_16 f 335%336
2 3 7 48
```

The initial result of the function must be the identity element for the primary function, which for catenation is an empty array of the appropriate shape—in the case of Sylvester's algorithm this is an empty vector.

An example using recursion

A good way to introduce recursion is by one of the oldest of all algorithms: the calculation of pi by approximating inscribed (and circumscribed) polygons.[44] The symbol pi (π) was chosen by William Jones (1706) because pi is the length of the perimeter of a circle of unit diameter. An inscribed hexagon has 6 sides each of length 0.5, which gives 3 as the first approximation.[45]

Doubling the sides of the hexagon gives a better approximation, and further doublings give still closer values. The secret is, therefore, to compute the length of a new chord from the length of an old one, which is not difficult to do once the theorem of Pythagoras is known. *CH* gives the new chord as a function of the old one.

```
     ∇Z←CH X
[1]    Z←(0.5×1-(1-X*2)*0.5)*0.5
[2]  ∇
```

For a circle of unit diameter, the first approximation is given by the perimeter of a hexagon whose sides are each equal to the radius, i.e., the approximation to pi is 3.

After 8 doublings (8 applications of *CH*), pi is given by:

```
      6×(2×2×2×2×2×2×2×2)×CH
            CH CH CH CH CH CH CH CH 0.5
3.14159
```

We have a notation (exponentiation) that allows us to abbreviate this to:

```
      6×(2*8)×CH CH CH CH CH
            CH CH CH CH 0.5
3.14159
```

With APL we can use recursion to effect successive applications of the function *CH*:

```
     ∇Z←N C X
[1]    ⍎(N=0)/'→0, 0⍴Z← X'
[2]    Z←(N-1) C CH X
[3]  ∇

     ∇Z←PI N
[1]    Z←6×(2*N)× N C 0.5
[2]  ∇
```

In direct definition these functions can be given more concisely:

```
      CH:(.5×1-(1-ω*2)*.5)*.5
      C:(α-1)C CHω: α=0 : ω

      PI:6×(2*ω)× ω C 0.5

      PI 8
3.14159
```

Because Iverson's J includes primitives for square root (%:), halve (-:), and square (*:), and a conjunction (dyadic operator) for raising a function to a power (^:), we have the following formulation:

```
     ch=. '%: -: 1- %: 1- *: y.' : ''

     6*(2^8)*ch ch ch ch ch ch ch ch 0.5
3.14159

     6*(2^8)*(ch^:8) 0.5
3.14159
```

Though the ancient Egyptians used *heap* as a general term for an unknown quantity,[46,47] Diophantus, a Greek mathematician in Alexandria about 300 AD, was probably the original inventor of an algebra using letters for unknown quantities.[48] Diophantus used the Greek capital letter *delta* (not for his own name!) for the word *power* ("*dynamis*"; compare "dynamo," "dynamic," and "dynamite"), which is therefore one of the oldest terms in mathematics.[14] Today we use a conjunction to raise a function to a power. The syntax brings out the parallelism between raising a number to a power and applying a function an equal number of times. The algorithm fails when the number of doublings is further increased.[49]

Hindu-Arabic numerals and zero

Hindu-Arabic numerals were introduced to the western world by Leonardo of Pisa (Fibonacci) in 1202 with these words: "*Novem figure Indorum he sunt 9 8 7 6 5 4 3 2 1. Cum his itaque nouem figuris, et cum hoc signo 0, quod arabic cephirum appellatur, scribitur quilibet numeros.*" [The nine numerals of the Indians are these: 9 8 7 6 5 4 3 2 1. With them and with this sign 0, which in Arabic is called cipher, any desired number can be written.[50]] (Slightly different in Reference 51.)

It was, however, far easier for most people to add and subtract with Roman numerals (or with Egyptian hieroglyphics for that matter), and this was sufficient for their needs. They also believed that, with the new system, accounts could be more easily falsified—for instance by changing zero into 6 or 9. Adoption of the new symbols was therefore very slow. The oldest known Hindu-Arabic numerals on a gravestone are dated 1371, and their earliest use on coins outside Italy was in 1424. They were not used on an English coin until 1551.[52] Even today Roman numerals are used for royalty. Clocks not powered by digital technology still commonly display old-style symbols on their dials.

As long as calculations were performed on *counters* or *boards* (see the etymology of *bank* and *bankrupt*) there was no need for a symbol to show an *empty* column. Menninger has some excellent sentences on the subject: "Zero is something that must be there to show that nothing is there, [for] only the abstract place-notation needs zero. Zero first liberated the digits from the counting board."[53]

Surely one of the most remarkable inscriptions in Europe[54] is: I · V^{c} · V . It records the date 1505 in symbols which, though Roman, are used with a positional significance unknown in Rome. The scribe "had heard about the new place-value system and now tried to find it in the Roman numerals. Since the meaning of the zero was still not clear to him, I V 0 V = 1505; at the critical point he yielded and retreated into the 'named' place-value notation."[55] He solved his problem by inserting a superscript letter *c* to identify the hundreds column (compare Sylvester's *locative symbols*). It is exciting to catch the conversion from the old way to the new as it was happening!

If it took so long for Hindu-Arabic numerals to make their way in the western world, we can hardly expect APL to be universally adopted in 25 years. But we can find encouragement in Menninger's words: "These ten symbols which today all peoples use to record numbers, symbolize the world-wide victory of an idea. There are few things on earth that are universal, and the universal customs which man has successfully established are fewer still. But this one boast he can make: the new Indian numerals are universal."[56]

One of the satisfactions in working with APL comes from its consistency and completeness, exemplified by its recognition of *identity elements*, i.e., arguments that, used with a dyadic function, give a result identical to the other argument. If at each iteration in a FORTRAN loop, we accumulate by adding to a variable named SUM, why must we set SUM to zero before entering the loop? The reason is that zero is the identity element for addition, as 1 is of multiplication. APL, being rich in scalar dyadic functions, needs more kinds of identity elements than other languages do.

Although the computation of pi by inscribed polygons is recursive, we did not accumulate intermediate results, but proceeded at once to the next approximation. On the other hand, Sylvester's algorithm for Egyptian unit-fractions constructs a vector, and the starting point must therefore be an empty vector.

We can calculate interest payments on a declining balance by following the same recursive paradigm.

Ordinary APL:

```
     ∇Z← A where W
[1]    Z←A
[2]  ∇
```

```
     ∇ZZ←N IB W;Z;B;R;I
[1]   ⍎(0>A←N-1)/'→0, 0ρZZ←0 3ρ0'
[2]   ZZ←Z,[0]A IB W[0 1],1↑Z←B,R,I
      where B←W[2]-R←W[1]-I←×/W[0 2]
[3]  ∇
```

Direct definition:

```
where:α:0:ω

IB:Z,[0]A IBω[0 1],1↑Z←B,R,I
   where B←ω[2]-R←ω[1]-I←×/ω[0 2]:
           0>A←α-1:0 3ρ0
```

where B = current balance; R = amount going to reduce principal; I = amount going to pay interest.

If the principal is $20,000, the interest is 10 percent, and the monthly payment is $1,000, the function *IB* computes a table for 12 months (numbers are rounded):

```
    12 IB 10 1000 20000÷1200 1 1
19167   833   167
18326   840   160
17479   847   153
16625   854   146
15763   861   139
14895   869   131
14019   876   124
13136   883   117
12245   891   109
11347   898   102
10442   905    95
 9529   913    87
```

In J's pure functional form, define *i*, *r*, and *b* as three forks:

```
   i=. 2&{ * {.
   r=. 1&{ - i
   b=. 2&{ - r

ib=.((b,r,i)@],
     <:@[ ib (2&{.,b)@])'
     (0 3&$ @ 0:) @.(=0:)
```

To understand the structure of this function, condense it as follows:

```
ib=.(f@], <:@[ ib g@])'h
      @.(=0:)
```

Read it thus:

To the result of function *f* of the right argument, catenate the item (row) resulting from the function *ib* with a decremented left argument, and a right argument computed by function *g* from the previous right argument. Function *h* gives the identity element for catenation of rows to a table with 3 columns. Terminate when the left argument is zero.

Because calculation of interest payments on a declining balance builds a table, we must start with 0 rows and 3 columns. Zero, then, is not enough; any language is incomplete if it fails to include different kinds of emptiness.

The identity element for matrix multiplication is the appropriately named *identity matrix*, first recognized by Cayley: "A matrix is not altered by its composition, either as first or second component matrix, with the matrix unity."[17] In the following example, the recursive function *MP* raises a matrix (left argument) to an integer power (right argument), and consequently requires the identity matrix of the same shape as the matrix argument.

Ordinary APL:

```
     ∇Z←M MP N;I
[1]   ⍎(N=0)/'→0, 0ρZ←I∘.=I←⍳1↑ρM'
[2]   Z←M+.×M MP N-1
[3]  ∇
```

More succinctly in direct definition:

```
MP:α+.×αMPω-1: ω=0 :I∘.=I←⍳1↑ρα

      M←3 3ρ⍳9
      M+.×M+.×M
180  234  288
558  720  882
936 1206 1476

      M MP 3
180  234  288
558  720  882
936 1206 1476
```

Zero seems to behave like the queen in chess; for is it not the most powerful piece on the board? Any number multiplied by zero is reduced to zero. But *emptiness* is more powerful still, because any number, including zero, is reduced to emptiness when multiplied by an empty vector. Emptiness is not, however, to be confused with *nothing*, which is the result of executing an empty vector. You cannot multiply a number by nothing—a *value error* results if you try. Shakespeare made the fool touch something profound in saying to the king without a throne: "Now thou art an O without a figure. I am better than thou art now; I am a Fool, thou art nothing."[57]

Unlike the play on words in Lewis Carroll's *Through the Looking Glass*, the distinctions between *zero*, *emptiness*, and *nothing* are not only useful but essential. The recursive APL functions already given include in a single line, *zero*, an *empty vector*, and (when the end condition obtains) *nothing*.

Logic

Because logic deals with two states, true and false, the mathematics of 0 and 1 is said to be logical. Propositions, or statements that may be judged true or false, are logical statements, and computers are logical machines because they manipulate binary digits (bits). The mathematics of logic began with Boole,[58-61] just at the time Sylvester introduced the term *matrix*. Jevons considered Boole's work to be, perhaps, "one of the most marvellous and admirable pieces of reasoning ever put together."[62] Bertrand Russell thought highly of Boole's work, going so far as to claim that "Pure mathematics was discovered by Boole in a work which he called 'The Land of Thought.' "[63]

"Let us conceive, then," wrote Boole, "of an Algebra in which the symbols x, y, z, etc. admit indifferently of the values of 0 and 1, and of these values alone."[64] Today we call a vector consisting of 1s and 0s a logical or Boolean vector, and Iverson notation, from its outset, used Boolean vectors to select from arrays, whether or not they were logical.[1] Where Boole used x(s) to stand for the selection of all the x s from subset s, Iverson used u/s in APL (or u#s in J), which is compression if u is Boolean and replication if it is not.

Because a computer's memory and registers can be described as arrays of 1s and 0s, we now recognize that Boole laid the foundation for the design and description of modern computers—which are logical machines. But to most of his contemporaries his work seemed of little significance. The obituary notice in *The Athenaeum* (December 17, 1864) dryly reported that "The Professor's principal works were 'An Investigation into the Laws of Thought,' and 'Differential Equations,' books which sought a very limited audience, and we believe found it."

The Oxford English Dictionary cites the use of *Boolian algebra* [sic] in 1895 and 1902, but however we spell it, the usage is questionable. As Sylvester emphasized, there is only one universal algebra, which must, of course, include logic: "I have also a great repugnance to being made to speak of Algebras in the plural; I would as lief acknowledge a plurality of Gods as of Algebras."[65] I am sure he would have approved of APL, which incorporates logical functions so that they can be used together with arithmetic functions in a single expression. For example, from Iverson:[66]

"A theorem is a proposition which is claimed to be universally true, i.e., to have the value 1 when applied to any element in the universe of discourse. For example, the proposition

$$((0=2|X)\wedge(0=3|X))\leq 0=6|X$$

is a theorem which may be verbalized in a variety of ways:

"X is divisible by 2 and X is divisible by 3 implies that X is divisible by 6.

"Any number divisible by both 2 and 3 is also divisible by 6.

"If X is divisible by both 2 and 3 then X is divisible by 6.

"Divisibility by 2 and 3 implies divisibility by 6."

According to John Venn (whose name is well known in connection with the diagrams that so effectively illustrate the meanings of *and*, *or*, and *not*), Jevons "was certainly the first to popularize the new conceptions of symbolic logic." The boldness, originality, and beauty of Boole's system fascinated him, and Jevons's book[67] was largely founded on Boole. Jevons, unlike Boole, emphasized the importance of the *inclusive or* and his symbol $(\cdot|\cdot)$ survives (though without the dots) in PL/I and in countless IBM technical manuals.

In 1865, Jevons completed construction of his reasoning machine, or logical abacus, adapted to show the workings of Boole's logic in a half mechanical manner, a full account of which was published by the Royal Society in 1870.[68] Mechanical devices had been designed by Napier, Pascal, Thomas of Colmar, and in Jevons's own time by Babbage, Stanhope,[69] and Smee,[70,71] but Jevons claimed that until the work of Boole, logic had remained substantially as molded by Aristotle 2200 years ago. De Morgan, whose *Formal Logic*[72] was published, by coincidence, on the same day as Boole's book,[58] pointed to the connection between two revealing facts: "logic

is the only science which has made no progress since the revival of letters; logic is the only science which has produced no growth of symbols." In my view APL is in the best tradition of Boole, De Morgan, Jevons, and Venn.[73]

One of the most striking features in Iverson's *A Programming Language* is his demonstration that "the generalized matrix product and the selection operations together provide an elegant formulation in several established areas of mathematics. A few examples will be chosen from two such areas, symbolic logic and matrix algebra."[74] Iverson proceeded to show how his notation leads to a natural extension of De Morgan's laws.[75]

De Morgan's law:

```
A ∧ B ↔ ~(~A) ∨ ~B
```

Iverson's extensions:

```
∧/U ↔ ~∨/~U
≠/U ↔ ~=/~U
```

In ordinary APL:

```
      U←? 5 4 3 ⍴ 2
      V←? 3 6 7 ⍴ 2

      ∧/, (U ≠.∧ V)=~ (~U) =.∨ (~V)
1
```

In J, the latest form of Iverson's notation, his 1962 example is executed as follows:

```
      u=.?5 4 3 $2
      v=.?3 6 7 $2

      (u~:/ .*.v)-:-.(-.u) =/ .+.(-.v)
1
```

where:

~: is NOT EQUAL; *. is AND; -: is MATCH; -. is NOT; and +. is OR.

In algebra a leading negative can be removed by changing the signs of all quantities in the expression that follows; in APL a leading NOT (~) can be removed by interchanging the pairs AND and OR, EQUALS and NOT-EQUALS, etc. In the following example both functions *F* and *G* remove redundant blanks from a string.

Ordinary APL:

```
      ∇Z←F S;U
[1]    Z←(~U∧1⌽U←S=' ')/S
[2]   ∇

      ∇Z←G S;U
[1]    Z←(U∨1⌽U←S≠' ')/S
[2]   ∇
```

Direct definition:

```
F:(~ U∧1⌽U←⍵=' ')/⍵
G:(U∨1⌽U←⍵≠' ')/⍵
```

APL continues to grow in power, and Iverson's final example,[76] written but not executable as +./ in APL, can be executed in J as follows.

Given:

```
A=. 1 3 2 0, 2 1 0 1,: 4 0 0 2
B=. 4 1, 0 3, 0 2,: 2 0
f=. ~:&0
h=. +/ @ #"0
```

Then:

```
(f A) +/ .h B
4 6
6 4
6 1
```

Iverson's generalized matrix product found immediate application in his formal description of indexed addressing on the IBM 7090 computer,[77] which in one line made clear what takes half a page of text in the *Principles of Operation* manual for that machine. There are, of course, many similar examples in Reference 78.

Arrays and locative symbols

APL is often referred to as the *array processing language*, and its power does to a great extent come from its ability to work with arrays directly, a feature of increasing importance as vector processors and parallel computing become available. When we specify a place by giving its latitude and longitude, or define a point on a scatter diagram by giving its X and Y coordinates, we intend that two numbers should be taken together to identify one object. This is the first step in thinking in terms of what Sylvester called *multiple quantity*.

Stevinus was the first to show how forces combine in the manner we know as the parallelogram of

forces.[79–81] The discovery is so important that Newton[82] stated it as Corollary I immediately after his Laws of Motion. Authors of modern textbooks often suggest that the rule for vector addition is quite arbitrary by saying that the sum of two vectors is defined to be a third vector whose components are given by the sum of the corresponding components of the given vectors. Such a statement disguises the fact that in the real world we observe that forces combine in this manner.

Many first encounter the word *vector* in Kepler's so-called Second Law of Planetary Motion: the radius vector sweeps out equal areas in equal times. Kepler's prodigious calculations are even more remarkable when we remember how few mathematical symbols were available—logarithms, and even the decimal point had not yet been invented.

Once Kepler had found a mathematical relationship that held throughout space, he looked for a deeper reason. Introducing the *Newtonian* concept of force into science, he claimed that a magnetic force (*anima motrix*) emanated from the sun and carried the planets in their orbits.[83]

Vector is the Latin word for a carrier, and it is used in medicine today in this sense. *Vector meus* is "my horse," and *vehicle*, *wagon*, *way*, and *convection* are from the same root. It was therefore an appropriate word for whatever it is that carries the planets in their orbits round the sun. I looked in vain for it in Kepler,[84] but Small[85] gives *radii vectores*. Harris, in 1704, defines *vector* to be "A line supposed to be drawn from any Planet moving round a Centre, or the Focus of an Ellipse, to that Centre or Focus, is by some writers of the New Astronomy, called the Vector; because 'tis that line by which the Planet seems to be carried round its Centre."[86]

A vector in two dimensions can be represented by a complex number (and vice versa). Wessel, a Norwegian surveyor, was the first to realize this, but his work, though published in 1799, was unrecognized until 1897. A modern geometric treatment of the addition and multiplication of complex numbers was given by Argand in 1806, but these ideas received little attention until Gauss took up the topic in 1831.

If complex numbers can represent points in a plane, it is natural to try to create hypercomplex numbers to represent points in three-dimensional space. Sir William Rowan Hamilton finally succeeded in doing this in 1843.[87]

In a long paper on "algebraic couples" written in 1837 Hamilton said: "In the THEORY OF SINGLE NUMBERS, the symbol $\sqrt{-1}$ is 'absurd' [it is an impossible root, or an imaginary number]; but in the THEORY OF COUPLES, the same symbol $\sqrt{-1}$ is 'significant' [i.e., it denotes a possible root, or a real couple]." What did he mean? I found the answer more clearly in Hamilton's own words than in modern textbooks.

Knowing that if you double a force you double the vector that represents it, Hamilton looked on *2 times* as the operator that doubles; it is a special case of what he called a *tensor*, an operator that stretches (not to be confused with the modern use of the word). In the same way *−1 times* is a reversor. Moreover if *$\sqrt{2}$ times* is applied twice it doubles; and if *$\sqrt{-1}$ times* is applied twice it reverses. Consequently *i times* (where i is $\sqrt{-1}$) is a versor, or operator that rotates a vector without changing its length; it is taken as producing a counter-clockwise rotation of 90 degrees. Application of *−2i times* would then be the composition of a rotation, a stretch, and a reversal. It is to Hamilton that we owe our terms *scalar* and *vector* (1846).

It seemed plausible that if couplets represent vectors in two dimensions, triplets would represent vectors in three dimensions, but after years of unsuccessful attempts, Hamilton realized, in a flash of genius, that a consistent algebra of triplets is impossible. Four terms (quaternions) are needed, shown in the example below:

complex: $a + bi$
$i^2 = {}^-1$

quaternion: $a + bi + cj + dk$
$i^2 = j^2 = k^2 = ijk = {}^-1$
$ij = -ji$

Quaternions are of interest to the pure mathematician because they do not obey the laws of ordinary arithmetic: multiplication of quaternions is associative but not commutative.

Hermann Grassmann (a German schoolmaster) worked on vector systems at about the same time as Hamilton, and it was Grassmann who, in 1862, gave us *inner* and *outer products*, analogous to the scalar and vector parts of Hamilton's multiplication of quaternions.[88–90]

All of Arthur Cayley's early papers were on, or used, determinants, and both he and Sylvester pub-

lished on the rotation of a solid body. These are all topics that led naturally to the algebra of matrices. A matrix can, as we know, be looked upon as an array of multidimensional vectors, and so it is interesting that in 1843, the year Hamilton discovered quaternions, Cayley published on "the Geometry of (n) dimensions." Work on matrices was almost bound to follow.

Cayley was much influenced by Hamilton and visited Hamilton in Dublin. Cayley wrote his first paper on quaternions in 1845 at the age of 24, and considered the quaternion theory to be "a generalization of the analysis which occurs in ordinary Algebra." Later the same year he wrote on "The octuple system of imaginaries," showing that consistent arithmetics exist for couples, quadruples (but not triplets), and eight-fold hypercomplex numbers. Two years later he demonstrated that "in the octuple system of imaginary quantities neither the commutative nor the distributive law holds."

In 1848 Cayley showed that the combined effect of two rotations could be represented as the product of two quaternions, and shortly afterwards Sylvester (in the year he introduced the term *matrix*) pointed out that any number of rotations can be represented by a single rotation about one axis. As we would now say: each rotation can be represented by a matrix, and the product of these matrices is a matrix completely describing the combined rotation, whose axis is an eigenvector of this matrix, and the angle of rotation can be found from the corresponding eigenvalue. By 1855 Cayley used matrix product (calling it the *composition* of matrices), and in his memoir of 1858 he wrote: "It will be seen that matrices comport themselves as single quantities; they may be added, multiplied, or compounded together, etc.: the law of the addition of matrices is precisely similar to that for the addition of ordinary algebraical quantities; as regards their multiplication (or composition), there is the peculiarity that matrices are not in general convertible; it is nevertheless possible to form the powers (positive or negative, integral or fractional) of a matrix . . . "[17] In this memoir he uses Sylvester's latent roots (eigenvalues), but without naming them.

Sylvester's paper, written in 1882, begins thus: "Professor Sylvester referred to the general question of representing the product of sums of two, four, or eight squares under the form of a like sum, and mentioned that Professor Cayley had been the first

Figure 8 Sylvester's locative symbols

$$\theta + 8h + 8t + 2u$$

is 1882

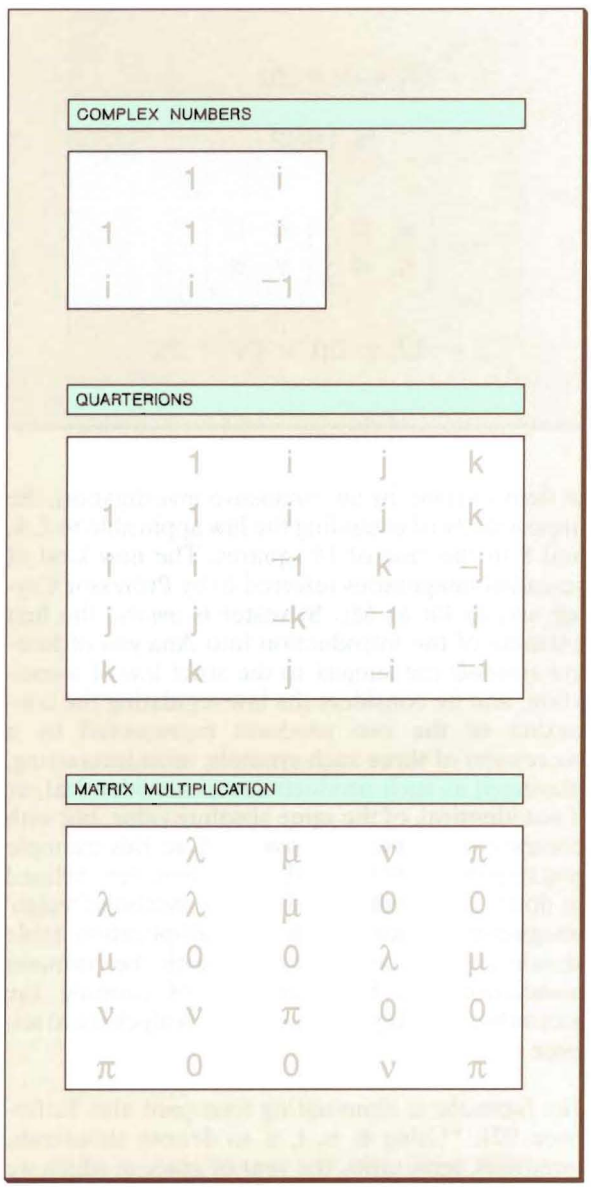
$$\rho = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \lambda & \mu \\ \nu & \pi \end{pmatrix}$$

$$\rho = a\lambda + b\mu + c\nu + d\pi$$

to demonstrate, by an exhaustive investigation, the impossibility of extending the law applicable to 2, 4, and 8 to the case of 16 squares. The new kind of so-called imaginaries referred to by Professor Cayley are, as far as Mr. Sylvester is aware, the first example of the introduction into Analysis of *locative symbols* not subject to the strict law of association, and he considers the law regulating the connexion of the two products represented by a succession of three such symbols, most interesting, inasmuch as such products are either identical, or if not identical, of the same absolute value, but with contrary signs: most persons, before this example had been brought forward, would have felt inclined to doubt the possibility of locative symbols ('vulgo' imaginary quantities) whose multiplication table should give results inconsistent with the common associative law, being capable of forming the groundwork of any real accession to algebraical science . . . . "[91]

His footnote is illuminating (compare also Reference 92): "Using θ, h, t, u to denote thousands, hundreds, tens, units, the year of grace in which we live may be represented by θ + 8h + 8t + 2u [—] θ, h, t, u, being locative symbols which it would be absurd to style 'imaginary quantities'; but they are as much entitled to that name as the i, j, k, or any like set of symbols—the only essential difference being that one set of symbols is limited, the other unlimited in number—and accordingly the law of combination of the one set is given by a finite and of the other an infinite 'multiplication table' . . . The 'locatives' indicate out of what 'basket,' so to say, the 'quantities' appearing in an analytical expres-

Figure 9 Sylvester's multiplication tables



sion are to be selected—the multiplication table determines the basket into which their product is to be thrown. . . . The whole analytical side of the theory of quaternions merges into a particular case of the general theory of *Multiple Algebra.* As far as the present writer is aware, Professor Cayley in his Memoir on Matrices (1858), was the first to recognize the parallelism between quaternions and matrices . . . "[91]

Sylvester's locative symbols and multiplication tables for complex numbers, quaternions, and matrix multiplication are given in Figures 8 and 9 (from References 18, 91, 93, 94). By this method of representation Sylvester states in 1884: "a matrix is robbed as it were of its areal dimensions and represented as a linear sum." Sylvester's 2 by 2 matrices *I*, *L*, *M*, and *N* are given in Figure 10, where the matrices, "construed as complex quantities, are a linear transformation of the ordinary quaternion system 1, i, j, k." As he said: "Every matrix of the second order may be regarded as representing a quaternion, and vice versa."

Sylvester's matrix identities given in Figure 10 can be demonstrated very concisely in Iverson's J, which supports complex numbers. The inner product is given by *p*, and *square* computes the product of a matrix with itself; *i* is $\sqrt{-1}$. One line suffices to express the identities. The *match* function is −:

```
i=.%:_1
p=.+/ .*
square=.p~

I=. 1 0,: 0 1
L=. (i,0),:0, -i
M=. 0 _1,: 1 0
N=. (0,-i),: -i,0

(<-I)-:&.> (square &.> L;M;N),
      <L p M p N
```

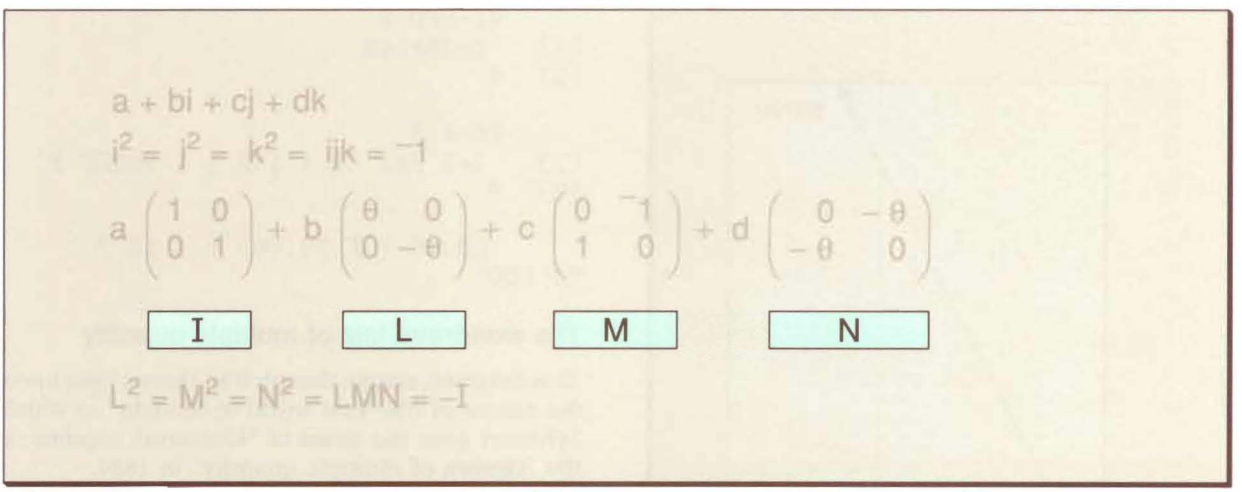
```
┌─┬─┬─┬─┐
│1│1│1│1│
└─┴─┴─┴─┘
```

These matrices, derived by Sylvester (see also References 71, 95) as an exercise in pure mathematics, are intimately connected to the Pauli spin matrices, which have central significance in relativistic quantum theory; they are also close to the *spinor transformation*,[96] to *basis quaternions*, and the *basis elements* of the 16-dimensional Clifford numbers,[97] whose algebraic properties can easily be demonstrated in APL.[98–100] The three Pauli matrices (σ_1, σ_2, and σ_3) describing the spin of an electron, together with all permutations of Pauli's identities, can be stated formally and executed. These are shown below in J with the numbers in square brackets from Pauli.

Given:

```
p=. +/ .*
i=. %: _1
```

Figure 10 Sylvester's 2 by 2 matrices



The matrices are:

```
      I=. 1 0,: 0 1

      s1=. 0 1,: 1 0                          [33.10]
      s2=. (0,-i),:(i,0)
      s3=. 1 0,: 0 _1
```

and the permutations are:

```
      z=. 0 1 2|.y=. s1;s2;s3

      I -:"2 p~"2 > y                         [33.9]
1 1 1

      f=. p-p~                                [33.11]
      g=. ({. f 1&{) -: (2*i)&* @ (2&{)
      1 -:"0 g "3 > z
1 1 1

      f=. p : - @ p~                          [33.12a]
      g=. {. (> @ f) 1&{
      h=. g -:"2 i&* @ (2&{)
      1 1 -:"1 h "3 > z
1 1 1

      f=. p+p~                                [33.12b]
      g=. {. f (1&{)
      (0 0,:0 0) -:"2 g "3 > z
1 1 1
```

In each of these identities, function *f* describes the essential relationship; functions *g* and *h* make it possible to test all "cyclical permutations of the indices."[98]

After Sylvester returned to England, the principal exponents of the New Algebra in the United States were Benjamin Peirce and J. Willard Gibbs. Sylvester called Peirce's 1870 memoir[95] "a work which may almost be entitled to take rank as the 'Principia' of the philosophical study of the laws of algebraical operation." Gibbs's address "On Multiple Algebra" to the Section of Mathematics and Astronomy of the American Association for the Advancement of Science is a classic. In it Gibbs wrote the following:

"The multiple quantities corresponding to concrete quantities such as ten apples or three miles are evidently such combinations as ten apples + seven oranges, three miles northwestward + five miles eastward, or six miles in a direction 50 degrees east of north . . . . But if we ask what it is in multiple algebra which corresponds to an abstract number like twelve, which is essentially an operator, which changes one mile into twelve miles, and $1,000 into $12,000, the most general answer would evidently be: an operator which will work changes as, for example, that of ten apples + seven oranges into fifty apples and 100 oranges, or that of one vector into another. If the operation is distributive, it may not inappropriately be called multiplication, and the result is par excellence the product of the operator and the operand. The sum of operators, quâ operators, is an operator which gives for the product the sum of the products given by the operators to be added. The product of two operators is an operator which is equivalent to the successive operations of the factors."[101]

Figure 11 Gibbs's example of transformation

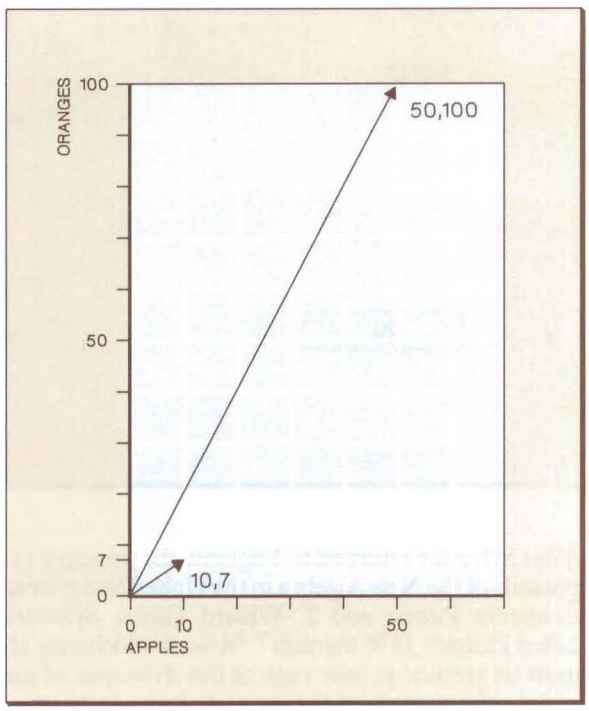


Figure 11 illustrates the problem Gibbs posed and makes the answer obvious. Although Gibbs did not turn to Hamilton, Sylvester, or Cayley for the solution, I betray their influence in Figure 12, where I separate the versor (as a rotation matrix) and the tensor (a scalar). The example can be worked as follows:

The transformation matrix (with tensor and versor composed):

```
      X← 50 100 ⍪ 10 ¯7,[¯0.5] 7 10
      N← (1 ¯1 × X),[¯0.5] ⌽X
      N+.× 10 7
 50 100
```

Isolate the tensor and determine the angle of rotation in degrees:

```
      ⎕← Y← (+/ X*2)*0.5
9.16

      (180÷○1)× ¯2 ¯1 ○ X÷Y
28.44 28.44
```

Confirm by composing the tensor and versor, where RFD is Radians From Degrees:

```
      ∇Z←RFD X
[1]    Z←○X÷180
[2]   ∇

      ∇Z←F X
[1]    Z←2 2⍴1 ¯1 1 1×2 1 1 2○RFD X
[2]   ∇

      (9.16 × F 28.44) +.× 10 7
50 100
```

The wondrous tale of multiple quantity

This example, simple though it is, throws light upon the nature of the "new world of thought" to which Sylvester gave the name of "Universal Algebra or the Algebra of multiple quantity" in 1884.

James Joseph Sylvester was born in 1814. In 1837 he completed his studies at Cambridge and published the first of his 342 papers. It was on crystallography. His next two papers were on the motion of fluids and rigid bodies—all topics of importance to my own subject of geology—and all amenable to matrix algebra. Some additional history can be found in Reference 102.

Sylvester,[103–106] the self-styled mathematical Adam, gave "more names (passed into general circulation) to the creatures of mathematical reason than all the other mathematicians of the age combined" (1888). In 1850, the year he was called to the bar, he introduced the term *matrix* for "a rectangular array of terms, out of which different systems of determinants may be engendered as from the womb of a common parent."[107,108] Sylvester introduced[109] the Greek letter *lambda* (λ) for the latent roots of a characteristic equation (his terms) in 1852—three-quarters of a century before the term *eigenvalue* was invented; and in 1853 he introduced the *inverse matrix*.[110]

In 1884, at the age of 70, he published his Lectures on the Principles of Universal Algebra, the "apotheosis of algebraical quantity," in the *American Journal of Mathematics*, which he himself founded and edited. His title reminds us that Newton used the term *universal arithmetic* for what we call *algebra*. Emphasizing the importance of matrices as multiple quantity, he speaks of a second birth of algebra, its *avatar* in a new and glorified form.[111] In the words of this enthusiast, who lived a century before APL was implemented: "A matrix of quadrate

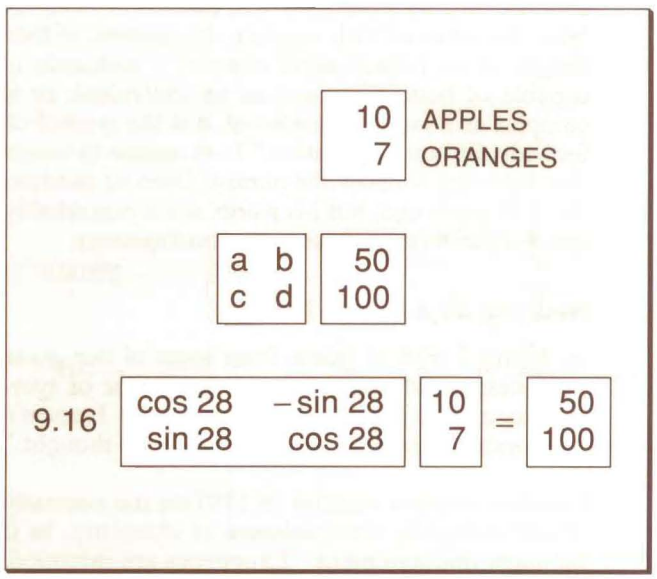
form ... emerges ... in a glorified shape—as an organism composed of discrete parts, but having an essential and undivisible unity as a whole of its own. ... The conception of multiple quantity rises upon the field of vision. ... [Matrix] dropped its provisional mantle, its aspect as a mere schema, and stood revealed as bona fide multiple quantity subject to all the affections and lending itself to all the operations of ordinary numerical quantity."

"This revolution," he continued, "was effected by a forcible injection into the subject of the concept of addition; that is, by choosing to regard matrices as susceptible to being added to one another; a notion as it seems to me, quite foreign to the idea of substitution, the nidus in which that of multiple quantity was laid, hatched and reared. This step was, as far as I know, first made by Cayley ... in his [immortal] Memoir on Matrices [1858], wherein he may be said to have laid the foundation-stone of the science of multiple quantity. That memoir indeed (it seems to me) may in truth be affirmed to have ushered in the reign of Algebra the 2nd; just as Algebra the 1st ... took its rise in Harriot's Artis Analyticae Praxis, published in 1631, ... exactly 250 years before I gave the first course of lectures ever delivered on Multinomial Quantity, in 1881, at the Johns Hopkins University."[18] References 112 to 115 add some additional information about Cayley.

If Sylvester were here today, what pleasure would he find in Iverson's notation, implemented even on our personal computers as an interactive language—this notation that encourages, and as it were expects, us to think in terms of arrays or multiple quantities, manipulating them as entities in the spirit of Sylvester's exhortations! That eloquent mathematician would be even more moved, I am sure, by boxed arrays (arrays of arrays), and array processors, which are APL machines.

A century ago both Sylvester and Gibbs urged us to think in terms of arrays. Most computer languages and what Backus called (perhaps unfairly) the Von Neumann bottleneck, force us, however, to work with scalars. Within the confines of a few pages, I have attempted to trace the development of notation and methods from hieroglyphics to APL. I have tried to show that APL is much more than yet another computer language; that its intellectual importance is great; and that (yet again using Sylvester's words) APL continues "The wondrous tale of Multiple Quantity."

Figure 12 Separation of versor and tensor



The story will, of course, never be completed. We have seen the recent introduction of two hitherto undefined phrases now called *hooks* and *forks*.[116] One example of each must suffice here.

`+/ % # y` computes the sum over the reciprocals of the tally of `y`, which is unlikely to be useful, whereas, if we unify the phrase, placing it in parentheses, it becomes a fork `(+/ % #) y` equivalent to `(+/y) % (#y)`, which computes the mean (or means over the leading axis if the rank exceeds 1).

`(- mean) y` is a hook, equivalent to `y - (mean y)`, which gives the deviations from the mean, a necessary step in computing variance.

It should be noted that when we define the phrase, as for example `mean=. +/ % #` the phrase is unified without requiring parentheses. The functions used above for the Pauli identities are examples of forks. The statistical examples above include hook (sums of cross products) and fork (correlation coefficients). The function for interest on a declining balance (*ib*) includes a train of five functions, three of which (*i*, *r*, *b*) are forks, and it ends with an interesting hook.

In a paper published in 1866 we find Sylvester writing on the subject of operators. "The force of the bracket [i.e., parentheses] explains itself. This wonderful symbol has the faculty of extending itself

without ambiguity to every possible development, however new, of mathematical language. It is susceptible only of a metaphysical definition as signifying the exercise, with regard to its content, of that faculty of the human mind whereby a multitude is capable of being regarded as an individual, or a complex as a monad. In a word, it is the symbol of individuality and unification." I am unable to assert that Sylvester foresaw the *phrasal forms* of modern APL 125 years ago, but his words seem remarkably apt in reference to these new developments.

Notation as a tool of thought

In ending I wish to quote from some of our great predecessors who appreciated the power of symbols as an aid to reasoning, or in Ken Iverson's memorable phrase, "notation as a tool of thought."

Lavoisier wrote a memoir in 1787 on the necessity of reforming the nomenclature of chemistry. In it he made this statement: "Languages are intended, not only to express by signs, as is commonly supposed, the ideas and images of the mind; but are also analytical methods, by the means of which, we advance from the known to the unknown, and to a certain degree in the manner of mathematicians. . . . Algebra is the analytical method by excellence [sic]; it has been invented to facilitate the operations of the understanding, and to render reasoning more concise, and to contract into a few lines what would have required whole pages of discussion; in fine, to lead, in a more agreeable and laconic method [*plus commode, plus prompte et plus sûre*], to the solution of the most complicated questions. Even a moment's reflection is sufficient to convince us that algebra is in fact a language: like all other languages it has its representative signs, its method and its grammar, if I may use the expression: thus an analytical method is a language; a language is an analytical method; and these two expressions are, in a certain respect synonimous [sic]."[117]

In 1821, Babbage, in his thought-provoking paper "On the Influence of Signs in Mathematical Reasoning," said: "The quantity of meaning compressed into small space by algebraic signs is a circumstance that facilitates the reasonings we are accustomed to carry on by their aid. The assumption of lines and figures to represent quantity and magnitude, was the method employed by the ancient geometers to present to the eye some picture by which the course of their reasonings might be traced: it was however necessary to fill up this outline by a tedious description, which in some instances even of no peculiar difficulty became nearly unintelligible, simply from its extreme length: the invention of algebra almost entirely removed this inconvenience, and presented to the eye a picture perfect in all its parts, disclosing at a glance, not merely the conclusion in which it terminated, but every stage of its progress. At first it appeared probable that this triumph of signs over words would have limits to its extent: a time it might be feared would arrive, when oppressed by the multitude of its productions, the language of signs would sink under the obscurity produced by its own multiplication. . . . Fortunately however such anticipations have proved unfounded.

"Examples of the power of a well-contrived notation to condense into small space a meaning which would—in ordinary language—require several lines, or even pages, can hardly have escaped the notice of most of my readers: in the calculus of functions, this condensation is carried to a far greater extent than in any other branch of analysis, and yet, instead of creating any obscurity, the expressions are far more readily understood than if they were written at length. . . . The power we possess by the aid of symbols of compressing into small compass the several steps of a chain of reasoning, whilst it contributes greatly to abridge the time which our enquiries would otherwise occupy, in difficult cases influences the accuracy of our conclusions: for from the distance which is sometimes interposed between the beginning and the end of a chain of reasoning, although the separate parts are sufficiently clear, the whole is often obscure. . . . The closer the succession between two ideas which the mind compares, provided those ideas are clearly perceived, the more accurate will be the judgement that results."[118]

"The advantage of selecting in our signs, those which have some resemblance to, or which from some circumstance are associated in the mind with the thing signified has scarcely been stated with sufficient force: the fatigue, from which such an arrangement saves the reader, is very advantageous to the more complete devotion of his attention to the subject examined; and the more complicated the subject, the more numerous the symbols and the less their arrangement is susceptible of symmetry, the more indispensable will such a system be found. This rule is by no means confined to the choice of the letters which represent quantity, but is meant to extend, when it is possible, to cases

where new arbitrary signs are invented to denote operators. . . . The more complicated the enquiries on which we enter, and the more numerous the quantities which it becomes necessary to represent symbolically, the more essentially necessary it will be found to assist the memory by contriving such signs as may immediately recall the thing which they are intended to represent."[119]

Sylvester, in 1877, said "It is the constant aim of the mathematician to reduce all his expressions to their lowest terms, to retrench every superfluous word and phrase, and to condense the Maximum of meaning into the Minimum of language."[120]

Whitehead, in 1911, claimed that "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. . . . By the aid of symbolism we can make transitions in reasoning almost mechanically by the eye, which would otherwise call into play the higher faculties of the brain. It is a profoundly erroneous truism, repeated by all copy-books and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them."[121]

Bertrand Russell said: "The great master of the art of formal reasoning, among men of our own day, is an Italian, Professor Peano, of the University of Turin. He has reduced the greater part of mathematics (and he or his followers will, in time, have reduced the whole), to strict symbolic form, in which there are no words at all."

In the first paragraph of his book in 1959, Russell wrote: "There is one major division in my philosophical work: in the years 1899–1900 I adopted the philosophy of logical atomism and the technique of Peano in mathematical logic. This was so great a revolution as to make my previous work, except such as purely mathematical, irrelevant to everything I did later. The change in these years was a revolution; subsequent changes have been of the nature of an evolution."[122]

And finally, Giuseppe Peano himself, in his paper on "The Importance of Symbols in Mathematics" in 1915 wrote: "The oldest symbols, which are also the most used today, are the digits used in arithmetic, which we learned about 1200 from the Arabs, and they from the Indians, who were using them about the year 400. The first advantage that one sees in the digits is their brevity. . . . Further reflection reveals that these symbols are not just shorthand, i.e., abbreviations of ordinary language, but constitute a new class of ideas. . . . The use of digits not only makes our expressions shorter, but makes arithmetical calculation essentially easier, and hence makes certain tasks possible, and certain results obtainable, which could not otherwise be the case in practice. For example, direct measure assigned to the number Pi, the ratio of the circumference of a circle its diameter, the value 3. . . .

"Archimedes, about 200 B.C., by inscribing and circumscribing polygons about a circle, or rather by calculating a sequence of square roots, using Greek digits, found Pi to within 1/500. The substitution of Indian digits for the Greek allowed Aryabhata, about the year 500, to extend the calculation to 4 decimal places, and allowed the European mathematicians of 1600 to carry the calculation out to 15 and then 32 places, still following Archimedes' model. Further progress, i.e., the calculation of 100 digits in 1700, and the modern calculation of 700, was due to the introduction of series.

"The same thing may be said for the symbols of algebra. . . . Algebraic equations are much shorter than their expression in ordinary language, are simpler, and clearer, and may be used in calculations. This is because algebraic symbols represent ideas and not words. . . . Algebraic symbols are much less numerous than the words they allow us to represent.

"The evolution of algebraic symbolism went like this: first, ordinary language; then, in Euclid, a technical language in which a one-to-one correspondence between ideas and words was established; and then the abbreviation of the words of the technical language, beginning about 1500 and done in various ways by different people, until finally one system of notation, that used by Newton, prevailed over the others.

"The use of algebraic symbols permits schoolchildren easily to solve problems which previously only great minds like Euclid and Diophantus could solve. . . . The symbols of logic too are not abbreviations of words, but represent ideas, and their principal utility is that they make reasoning easier. All those who use logical symbolism attest to this."[123]

Concluding remarks

A progression of great thinkers has moved the human race towards the adoption, first of an economical and efficient number system containing zero and based on place value, and then of a universal algebra, APL, which operates on arrays or multiple quantities, and is totally devoid of words.

There have also been those who resisted the inevitable progress, who found it difficult to adopt new and improved tools for thought. In our own time we hear appeals to revert from this high intellectual level and use English words, and to submit to the tyranny of scalars, as if Sylvester's eloquence a century ago had fallen on deaf ears.

Unlike its predecessors, APL is an executable notation. APL represents, in a phrase used by Babbage, the "triumph of symbols over words." As so many of our distinguished predecessors predicted, it makes reasoning easier. APL is the result of brilliant insight, careful thought, and hard work through at least 5000 years. Iverson is the latest in a succession that includes Peano, Sylvester, Cayley, De Morgan, Boole, Newton, Leibniz, Napier, Stevinus, Fibonacci, Diophantus, and the unknown Egyptian whose work was copied by Ahmes the scribe.

In 1866 Sylvester proclaimed that: "To attain clearness of conception, the first condition is 'language,' the second 'language,' the third 'language'—Protean speech—the child and parent of thought."[124]

In reflecting on the significance of APL I have adopted a historical approach. Having done so I find that Sylvester had something to say on that subject also. The occasion was his Presidential Address to the British Association[125] in 1869 when he said: "the relation of master and pupil is acknowledged as a spiritual and lifelong tie, connecting successive generations of great thinkers with each other in an unbroken chain."

We think in a different way because of APL.

Acknowledgments

This paper was based on a series of lectures sponsored by ACM and IBM. Lou Solheim and Tom Olsen, Karsten Manufacturing Company, provided a transcript of my address to ACM's APL83; I. P. Sharp Associates provided a transcript of my address to the 1982 ACM APL users meeting; and Ed Shaw provided the transcript of my address to an ACM SIGAPL meeting in New York. Jon McGrew and Jay Friedman, IBM Corporation, went to great pains to aid in the preparation of the paper for inclusion in the *IBM Systems Journal*. I am indebted to Kenneth E. Iverson and Adin Falkoff for more than 20 years of stimulation, criticism, and advice, and to the late William J. Bergquist for showing me that Iverson's notation had been implemented as executable APL.

Cited references and notes

1. K. E. Iverson, "The Description of Finite Sequential Processes," *Proceedings of a Conference on Information Theory*, Colin Cherry and Willis Jackson, Editors, Imperial College, London (August 1960), pp. 447–457.
2. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962).
3. A. N. Whitehead, *An Introduction to Mathematics*, Home University Library, New York and London (1911).
4. A. C. Aiken, *The Case Against Decimalisation*, Oliver and Boyd, Edinburgh (1962).
5. R. J. Gillings, *Mathematics in the Time of the Pharaohs*, M.I.T. Press, Cambridge, MA (1972). Reprinted by Dover Publications Inc., New York (1982), p. 16.
6. F. Klein, *Elementary Mathematics from an Advanced Standpoint*, Third Edition first published in 1924–1925; English translation, 2 volumes, published by Dover Publications Inc., New York (1939).
7. F. Cajori, *A History of Mathematical Notations*, Vol. 1, The Open Court Publishing Company, La Salle, IL (1951), first published in 1928; Vol. 2, The Open Court Publishing Company, La Salle, IL (1952), first published in 1929.
8. E. E. McDonnell, "The Caret and Stick Functions," *APL Quote Quad* **8**, No. 4, 35–39 (June 1978).
9. F. Cajori, *op. cit.*, Vol. 1, pp. 230–231.
10. M. Kline, *Mathematical Thought from Ancient to Modern Times*, Oxford University Press, New York (1972), p. 262.
11. Contemporary publications of early mathematicians are difficult to find. I consulted Oughtred's books in the British Library and the National Library of Scotland. Instead of giving citations in the usual form, it is more useful to recommend the following two sources: A. De Morgan, *Arithmetical Books from the Invention of Printing to the Present Time. Being notices of a large number of works drawn up from actual inspection*, Taylor and Walton, London (1847), republished, with a biographical introduction by A. R. Hall, London (1967); and Reference 7.
12. M. Kline, *op cit.*, p. 988.
13. D. E. Smith, *Mathematics*, Our Debt to Greece and Rome Series, Marshall Jones Company, Boston, MA (1923).
14. H. W. Turnbull, *The Great Mathematicians*, Methuen and Company, London (1929). Reprinted in *The World of Mathematics*, Vol. 1, James R. Newman, Editor, Simon and Schuster, New York (1956), p. 115.
15. D. E. Smith, *History of Mathematics*, Vol. 2, first published in 1925, Dover Publications Inc., New York (1958), pp. 427–431.

16. S. Stevin or Stevinus (1548–1620), *La Thiende*, 1585. French translation, *La Disme enseignant facilement expédier par Nombres Entiers sans rompuz tous Comptes se rencontrans aux Affaires des Hommes* [the art of decimal arithmetic made easy: the use of whole numbers to perform quickly all business calculations]. English version *The Art of Tenths, or Decimall Arithmeticke* [sic] . . . *invented by Simon Stevin*, 1608 (compare the etymologies of *dime* and *tithe*).
17. A. Cayley, "A Memoir on the Theory of Matrices," Royal Society of London, *Philosophical Transactions* **148**, 17–37 (1858). Reprinted in *Collected Mathematical Papers* **2**, No. 152 (1889).
18. J. J. Sylvester, "On the Inverse and Negative Powers of a Matrix," Lectures on the Principles of Universal Algebra, *American Journal of Mathematics* **6**, 270–286 (1884).
19. F. Cajori, *op. cit.*, Vol. 2, pp. 15–29.
20. K. Menninger, *Number Words and Number Symbols*, German edition (1957), English translation, The M.I.T. Press, Cambridge, MA (1969), pp. 291, 353, 358; *The Treasury of Mathematics*, Henrietta O. Midonick, Editor, Philosophical Library, New York (1965).
21. F. Cajori, *op. cit.*, Vol. 1, pp. 366–369.
22. A. Gittleman, *History of Mathematics*, Charles E. Merrill Publishing Company, Columbus, OH (1975), pp. 141–149.
23. J. W. L. Glaisher, "Logarithms and Computation," in *Napier Tercentenary Memorial Volume*, Cargill Gilston Knott, Editor, London (1915), pp. 63–80.
24. P. E. B. Jourdain, "The Nature of Mathematics," reprinted in *The World of Mathematics*, Vol. 1, James R. Newman, Editor, Simon and Schuster, New York (1956), p. 23.
25. J. R. Newman, *The World of Mathematics*, Vol. 3, James R. Newman, Editor, Simon and Schuster, New York (1956), p. 1856.
26. H. T. Pledge, *Science Since 1500: A Short History of Mathematics, Physics, Chemistry, Biology*, H. M. Stationery Office, London (1939).
27. K. E. Iverson, *Elementary Analysis*, APL Press, Swarthmore, PA (1976), pp. 140–158.
28. H. Hellerman, *Digital Computer System Principles*, McGraw-Hill Book Company, Inc., New York, 1st Edition (1967); 2nd Edition (1973), p. 53.
29. K. E. Iverson, "A Personal View of APL," *IBM Systems Journal* **30**, No. 4, 582–593 (1991, this issue).
30. R. K. W. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney, "APL\?," *APL90 Conference Proceedings*, Copenhagen, Denmark (August 1990); *APL Quote Quad* **20**, No. 4, 192–200 (July 1990).
31. K. E. Iverson, *The ISI Dictionary of J*, Version 3.3, Iverson Software Inc., Toronto (1991).
32. The Rosetta stone (now in the British Museum) is a basalt slab with inscriptions in three notations: (1) hieroglyphics, (2) demotic, and (3) Greek, which provided the key to deciphering hieroglyphics. Like the Rosetta stone, my examples of (1) ordinary APL, (2) direct definition, and (3) J, provide an opportunity for those familiar with one particular notation to decipher others.
33. C. Babbage, "On the Influence of Signs in Mathematical Reasoning," Cambridge Philosophical Society, *Transactions* **2**, 333 (1827), read December 16, 1821.
34. R. Recorde, "The Ground of Artes" (1540), later edition, 1646. Reprinted in part in *The World of Mathematics*, Vol. 1, James R. Newman, Editor, Simon and Schuster, New York (1956), pp. 210–217.
35. K. Menninger, *op. cit.*, p. 426.
36. J. J. Sylvester, "On Staudt's Theorems Concerning the Contents of Polygons and Polyhedrons," with a note on a new and resembling class of theorems, *Philosophical Magazine* **4**, 383 (1852).
37. K. E. Iverson, *APL in Exposition*, Technical Report 320-3010, IBM Corporation (1972). Reprinted by APL Press, Swarthmore, PA (1976), pp. 19–23.
38. J. R. Newman, "The Rhind Papyrus," in *The World of Mathematics*, Vol. 1, James R. Newman, Editor, Simon and Schuster, New York (1956), pp. 169–178.
39. A. B. Chace, *The Rhind Mathematical Papyrus*, translation and commentary, The National Council of Teachers of Mathematics, Reston, VA (1979).
40. *The Rhind Mathematical Papyrus*, T. E. Peet, Editor, British Museum, London (1923).
41. R. J. Gillings, *op. cit.*, p. 104.
42. R. J. Gillings, *op. cit.*, pp. 104–127.
43. J. J. Sylvester, "On a Point in the Theory of Vulgar Fractions," *American Journal of Mathematics* **3**, 332–335, 388–389 (1880).
44. K. E. Iverson, *Elementary Functions: An Algorithmic Treatment*, Science Research Associates, Inc., Chicago (1966), pp. 15–16.
45. P. Beckmann, *A History of PI*, The Golem Press, St. Martins Press, New York (1971).
46. R. J. Gillings, *op. cit.*, p. 157.
47. P. E. B. Jourdain, *op. cit.*, p. 12.
48. P. E. B. Jourdain, *op. cit.*, p. 16.
49. G. Forsythe, "Pitfalls in Computation," *American Mathematical Monthly* **77**, No. 9, 931–956 (1970).
50. K. Menninger, *op. cit.*, p. 425.
51. D. E. Smith, *History of Mathematics*, Vol. 1, p. 216; Vol. 2, p. 71, footnotes, Dover Publications Inc., New York (1958).
52. F. Cajori, *op. cit.*, Vol. 1, p. 50.
53. K. Menninger, *op. cit.*, pp. 391–392, 398, 400.
54. *Ibid.*, pp. 285, 392, 400.
55. *Ibid.*, p. 392.
56. *Ibid.*, p. 391.
57. W. Shakespeare, *Lear*, Act 1, Scene 4.
58. G. Boole, *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning*, published in Cambridge and London (1847). Reprinted by Basil Blackwell, Oxford (1948). Also in G. Boole, *Studies in Logic and Probabilities*, with notes and additions, Watts and Company, London (1952), pp. 49–124.
59. G. Boole, *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*, Walton and Maberly, London (1854). Reprinted by Dover Publications Inc., New York (1953). Also reprinted by Peter Smith and by The Open Court Publishing Company, La Salle, IL.
60. A. De Morgan, "On the Symbols of Logic," Cambridge Philosophical Society, *Transactions* **9**, Part 1 (1850).
61. R. Feys and F. B. Fitch, *Dictionary of Symbols of Mathematical Logic*, North-Holland Publishing Company, Amsterdam (1969).
62. G. Boole, *Studies in Logic and Probability*, Watts and Co., London (1952), Appendix A, p. 471.
63. W. Kneale, "Boole and the Revival of Logic," *Mind* **57** (1948), pp. 149–175. Cited by J. R. Newman in *The World of Mathematics*, Vol. 3, James R. Newman, Editor, Simon and Schuster, New York (1956), pp. 1853–1854.
64. G. Boole, see Reference 58, p. 37.

65. J. J. Sylvester, "A Word on Nonions," *Johns Hopkins University Circulars* **1**, 241–242 (1882); **2**, 46 (1883).
66. K. E. Iverson, see Reference 37, p. 32.
67. W. S. Jevons, *Pure Logic, or the Logic of Quality Apart from Quantity*, London (1864).
68. W. S. Jevons, "On the Mathematical Performance of Logical Inference," Royal Society, *Philosophical Transactions* **160**, 497–518 (1870).
69. R. Harley, "The Stanhope Demonstrator, an Instrument for Performing Logical Operations," *Mind* **4**, 192–210 (1879).
70. A. Smee, *The Process of Thought Adapted to Words and Language, Together with a Description of the Relational and Differential Machines*, Longman, Brown, Green, and Longmans, London (1851).
71. C. S. Peirce, "Logical Machines," *American Journal of Psychology* **1**, 165–170 (1887).
72. A. De Morgan, *Formal Logic: or, the Calculus of Inference Necessary and Probable*, first published in London in 1847, A. E. Taylor, Editor, reprinted by Open Court Company, London (1926).
73. Sylvester had not only been a colleague of De Morgan's, but at the age of 13 had been De Morgan's pupil. He was the second person to be awarded the De Morgan medal (1887). The first was Cayley (1884). Cayley received a Royal Medal from the Royal Society in 1859, as Sylvester did in 1861. Sylvester received the Copley Medal in 1880; it is the highest honor possible from the Royal Society.
74. K. E. Iverson, see Reference 2, pp. 23–25.
75. M. Kline, *op. cit.*, p. 1189.
76. K. E. Iverson, *op. cit.*, p. 24.
77. K. E. Iverson, *op. cit.*, p. 73.
78. A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," *IBM Systems Journal* **3**, No. 3, 198–263 (1964).
79. S. Stevinus, *Statics and Hydrostatics* (1586).
80. W. W. Rouse Ball, *A Short Account of the History of Mathematics*, 4th Edition originally published in 1908, Dover Publications Inc., New York (1960), pp. 245–246.
81. M. Jammer, *Concepts of Force: A Study in the Foundations of Dynamics*, Harvard University Press, Cambridge, MA (1957). Reprinted by Harper Torchbooks, New York (1962), pp. 123–132.
82. I. Newton, *Sir Isaac Newton's Mathematical Principles of Natural Philosophy and His System of the World*, translated by Andres Motte in 1729. Revised and edited by Florian Cajori, University of California, Berkeley, CA (1946).
83. M. Jammer, *op. cit.*, pp. 85–93.
84. J. Kepler, *Astronomia Nova ... De Motibus Stellae Martis* (1609).
85. R. Small, *An Account of the Astronomical Discoveries of Kepler: Including an Historical Review of the Systems Which Had Successively Prevailed Before His Time*, J. Mawman, London (1804). Reprinted by University of Wisconsin Press, Madison, WI (1963), p. 198.
86. Harris, *Universal Dictionary of the Arts and Sciences* (1704).
87. While still an undergraduate, he was appointed to the Chair of Astronomy in Dublin, soon afterwards becoming Astronomer Royal of Ireland. Schrödinger called him "one of the greatest men of science the world has produced," and Whittaker said that "after Isaac Newton, the greatest mathematician of the English-speaking world is William Rowan Hamilton."
88. E. W. Hyde, *Grassmann's Space Analysis*, Mathematical Monograph, No. 6, 4th Edition, John Wiley & Sons, New York (1906).
89. F. Kline, *op. cit.*, Vol. 2, Chapters 2 and 3.
90. M. J. Crowe, *A History of Vector Analysis: The Evolution of the Idea of a Vectorial System*, Chapter 3, University of Notre Dame Press, South Bend, IN (1967).
91. J. J. Sylvester, "On the 8-Square Imaginaries," *Johns Hopkins University Circulars* **1**, 203 (1882).
92. K. Menninger, *op. cit.*, pp. 53–54.
93. J. J. Sylvester, "On the Involution and Evolution of Quaternions," *Philosophical Magazine* **16**, 394–396 (1883).
94. J. J. Sylvester, "Sur les Quantités formant un Groupe de Nonions analogues aux Quaternions de Hamilton," 2nd paper, *Comptes Rendus* **98**, 273–276, 471–475 (1884).
95. B. Peirce, *Linear Associative Algebra*, memoir read before the National Academy of Sciences in Washington, 1870. Reprinted with notes and addenda by C. S. Peirce, in *American Journal of Mathematics* **4**, 97–229 (1881).
96. C. W. Misner, K. S. Thorne, and J. A. Wheeler, "In Gravitation: Chapter 41," *Spinors*, W. H. Freeman and Company, San Francisco, CA (1973).
97. A. Kyrala, *Theoretical Physics: Applications of Vectors, Matrices, Tensors, and Quaternions*, W. B. Saunders Company, Philadelphia, PA and London (1967).
98. W. Pauli, "Pauli Lectures on Physics," *Wave Mechanics* **5**, English translation, Charles P. Enz, Editor, The MIT Press, Cambridge, MA and London, p. 158.
99. P. A. M. Dirac, *The Principles of Quantum Mechanics*, Oxford University Press (1935), pp. 67–70; 4th Edition (1958), pp. 149–151.
100. C. W. Misner, *op. cit.*, pp. 1135–1158.
101. J. W. Gibbs, "On Multiple Algebra," address before the Section of Mathematics and Astronomy of the American Association for the Advancement of Science by the Vice President, American Association for the Advancement of Science, *Proceedings* **35**, 37–66 (1886). Reprinted in *The Scientific Papers of J. Willard Gibbs, Ph.D., LL.D.*, Vol. 2, Dover Publications Inc., New York (1961).
102. In 1839, at the age of 25, Sylvester was elected a Fellow of the Royal Society. Although, in his own phrase, he was "one of the first holding the faith in which the Founder of Christianity was educated to compete for high honours in the Mathematical Tripos at Cambridge," he could not obtain his B.A. degree until 1872, after all religious tests had been abolished. At different times he was Professor of Physics in London, where he was a colleague of De Morgan's; Professor of Mathematics at the University of Virginia, where he left in haste after successfully defending himself with a sword-cane against the brother of a student whose work he had criticized; and Professor at the Royal Military Academy.
103. The currently popular movement that enjoys "debunking history and toppling eminent Victorians" has not spared Sylvester and Cayley. Hawkins (see Reference 104) says that "the significance of Cayley's memoir on matrices of 1858 has been grossly exaggerated." Sylvester is not even mentioned. Those interested may, however, consult References 105 and 106.
104. T. Hawkins, "The Theory of Matrices in the 19th Century," International Congress of Mathematicians, Vancouver, 1974, *Canadian Mathematical Congress* (1975), pp. 561–570.
105. A. Cayley, *Collected Mathematical Papers*, 13 volumes, Cambridge University Press, London (1889).

106. J. J. Sylvester, *Collected Mathematical Papers of James Joseph Sylvester*, with index and biographical notice by H. F. Baker, Editor, 4 volumes, Cambridge University Press, London (1904–1912).
107. J. J. Sylvester, additions to the articles "On a New Class of Theorems," and "On Pascal's Theorem," *Philosophical Magazine* **37**, 363–370 (1850).
108. J. J. Sylvester, "On the Relation Between the Minor Determinants of Linearly Equivalent Quadratic Functions," *Philosophical Magazine* **1**, 295–305 (1851).
109. J. J. Sylvester, "A Demonstration of the Theorem That Every Homogeneous Quadratic Polynomial Is Reducible by Real Orthogonal Substitutions to the Form of a Sum of Positive and Negative Squares," *Philosophical Magazine* **4**, 138–142 (1852).
110. J. J. Sylvester, "On a Theory of the Syzygetic Relations of Two Algebraic Functions," Royal Society of London, *Philosophical Transactions* **143**, Part III, 407–548 (1853).
111. J. J. Sylvester, "The Genesis of an Idea, or Story of a Discovery Relating to Equations in Multiple Quantity," *Nature* **31**, 35–36 (1884).
112. In his publications in two continents (and in France) Sylvester made many references to the memoir by Cayley (see Reference 17). With every reference to Cayley he pays the highest tribute. He refers to the memoir as "le beau Mémoire" (Reference 113), "his great paper on Matrices" (Reference 114), "Cayley's immortal Memoir" (Reference 115), and "Professor Cayley's ever-memorable paper on matrices. This paper constitutes a second birth of Algebra, its avatar in a new and glorified form" (Reference 111).
113. J. J. Sylvester, "Sur les Quantités formant un Groupe de Nonions analogues aux Quaternions de Hamilton," 1st paper, *Comptes Rendus* **97**, 1336–1340 (1883).
114. J. J. Sylvester, "On Quaternions, Nonions, Sedenions, etc.," *Johns Hopkins University Circulars* **3**, 33, 34, 57 (1884).
115. J. J. Sylvester, see Reference 18, pp. 270–286.
116. K. E. Iverson and E. E. McDonnell, "Phrasal Forms," *APL Quote Quad* **19**, No. 4, 197–199 (1989).
117. A. Lavoisier, *Méthode de Nomenclature Chimique: Nomenclature Chimique*, Paris (1787), pp. 1–25. *Chymical Nomenclature: A Memoir on the Necessity of Reforming and Bringing to Perfection the Nomenclature of Chymistry* [sic]; read to the Public Assembly of the Royal Academy of Sciences in Paris on the 18th of April, 1787, Edinburgh (1787), pp. 1–18.
118. C. Babbage, *op. cit.*, pp. 331–332.
119. *Ibid.*, pp. 370–371.
120. J. J. Sylvester, "Address on Commemoration Day at Johns Hopkins University, 22 February 1877," in *Collected Mathematical Papers* **3**, Number 10 (1877).
121. A. N. Whitehead, *op. cit.*, p. 59.
122. B. Russell, *My Philosophical Development*, George Allen and Unwin, Ltd., London (1959).
123. G. Peano, "The Importance of Symbols in Mathematics," originally published in Italian, *Scientia* **18**, 165–173 (1915); English translation in *Selected Works of Giuseppe Peano*, with a biographical sketch and bibliography by Hubert C. Kennedy; George Allen and Unwin, Ltd., London (1973), pp. 227–234.
124. J. J. Sylvester, "Note on the Properties of the Test Operators Which Occur in the Calculus of Invariants . . .," *Philosophical Magazine* **32**, 461–472 (1866).
125. J. J. Sylvester, *Presidential Address to Section "A" of the British Association*, Exeter British Association Report (1869), pp. 1–9.

General references

C. N. Anderson, *The Fertile Crescent: Travels in the Footsteps of Ancient Science*, Sylvester Press, Fort Lauderdale, FL, First Edition (1968), Second Edition (1972).

Anonymous, Obituary Notices: G. Boole, Royal Society, *Proceedings* **15**, vi–xi (1867).

Anonymous, Obituary Notices: W. S. Jevons, Royal Society, *Proceedings* **35**, i–xii (1883).

R. C. Archibald, "Mathematics Before the Greeks," *Science* **71**, 109–121, 342 (1930); **72**, 36 (1930).

J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," 1977 Turing Award Lecture, *Communications of the ACM* **21**, No. 8, 613–641 (1978).

E. T. Bell, *Men of Mathematics*, Simon and Schuster, New York, and Victor Gollancz, London (1937); Cayley and Sylvester, *Invariant Twins*, reprinted in *The World of Mathematics*, Vol. 1, J. R. Newman, Editor, Simon and Schuster, New York (1956), pp. 341–365.

M. Black, *The Nature of Mathematics*, with bibliography of symbolic logic, New York (1934).

C. B. Boyer, *A History of Mathematics*, John Wiley & Sons, Inc., New York (1968).

R. C. Buck, "Sherlock Holmes in Babylon," *The American Mathematical Monthly* **87**, No. 5, 335–345 (1980).

Sir E. A. Wallis Budge, *An Egyptian Hieroglyphic Dictionary*, Dover Publications Inc., New York, first published in 1920.

Sir E. A. Wallis Budge, *Egyptian Language: Easy Lessons in Egyptian Hieroglyphics*, Dover Publications Inc., New York, first published in 1910.

F. Cajori, "Algebra in Napier's Day and Alleged Prior Inventions of Logarithms," in *Napier Tercentenary Memorial Volume*, C. G. Knott, Editor, London (1915), pp. 93–109.

F. Cajori, *A History of Elementary Mathematics*, Macmillan Publishing Company, New York (1930).

F. Cajori, *A History of Mathematics*, Macmillan Publishing Company, New York (1919).

F. Cajori, "History of the Exponential and Logarithmic Concepts," *American Mathematical Monthly* **20** (1913).

F. Cajori, *William Oughtred: A Great Seventeenth-Century Teacher of Mathematics*, The Open Court Publishing Company, La Salle, IL (1916).

P. Carruthers, "Introduction to Unitary Symmetry"; Chapter 1, Angular Momentum and Isospin, *Interscience Tracts on Physics and Astronomy*, No. 27, Interscience, John Wiley & Sons, Inc., New York (1966).

H. S. Carslaw, "The Discovery of Logarithms by Napier," *The Mathematical Gazette* **8**, 76–84, 115–119 (1915).

M. Caspar, *Kepler*, Abelard-Schuman, London and New York (1959).

W. K. Clifford, *Application of Grassmann's Extensive Algebra*, Vol. 1 (1878).

M. R. Cohen and E. Nagel, *An Introduction to Logic and Scientific Method*, New York (1934).

A. W. Conway, "Quaternions and Matrices," Quaternion Centenary Celebration, 8th November, 1943, Royal Irish Academy, *Proceedings* **A50**, 98–103 (1945).

M. P. Crosland, *Historical Studies in the Language of Chemistry*,

first published in 1962, Dover Publications Inc., New York (1978).

B. Datta and A. N. Singh, *History of Hindu Mathematics: A Source Book*, Asian Publishing House, Bombay, London, and New York, Part 1 (1935); Part 2 (1938); single volume (1962).

A. De Morgan, *Syllabus of a Proposed System of Logic*, Maberley, London (1860). Reprinted in *On the Syllogism and Other Logical Writings by Augustus De Morgan*, edited with an introduction by P. Heath, Routledge and Kegan Paul, London (1966).

S. E. De Morgan, *Memoir of Augustus De Morgan*, Longmans, Green & Co., London (1882).

L. E. Dickson, "Linear Algebras," *Cambridge Tracts in Mathematics and Mathematical Physics*, No. 16 (1914), Hafner Publishing Company, New York; undated reprint.

Dictionary of Scientific Biography, 14 volumes, C. C. Gillispie, Editor, Charles Scribner's Sons, New York (1970–1976).

W. J. Dobbs, "The Teaching of Indices and Logarithms," *The Mathematical Gazette* **8**, 119–125 (1915).

J. M. Dubbey, *The Mathematical Work of Charles Babbage*, Cambridge University Press (1978).

Elements of Quaternions by the Late Sir William Rowan Hamilton, W. E. Hamilton, Editor, London (1866); Second Edition, 2 volumes, C. J. Joly, Editor, Longmans, Green & Co., London (1901).

H. Eves, *Introduction to the History of Mathematics*, first published in 1953, 4th Edition, Holt, Rinehart and Winston, New York (1976).

G. Flegg, *Numbers: Their History and Meaning*, Schocken Books, New York (1983).

H. G. Forder, *The Calculus of Extension*, Cambridge University Press, Cambridge (1941), reprinted by Chelsea (1960).

S. Gandz, "Studies in Babylonian Mathematics," *Osiris* **8**, 12–40 (1948).

M. Gardner, *Logic Machines and Diagrams*, McGraw-Hill Book Company, Inc., New York (1958).

J. W. Gibbs, *Elements of Vector Analysis*, privately printed, New Haven, CT (1881 and 1884).

J. W. Gibbs, "On the Role of Quaternions in the Algebra of Vectors," *Nature* **43**, 511–513 (1891).

J. W. Gibbs, "Quaternions and the 'Ausdehnungslehre,'" *Nature* **44**, 79–82 (1891).

J. W. Gibbs, *The Scientific Papers of J. Willard Gibbs, Ph.D., LL.D.*, Vol. 2—*Dynamics, Vector Analysis and Multiple Algebra, etc.*, Longmans, Green & Co., London (1906 and 1931); Dover Publications Inc., New York (1961).

J. W. Gibbs, *Vector Analysis*, E. B. Wilson, Editor, Charles Scribner's Sons, New York (1902).

J. W. L. Glaisher, "On the Early History of the Signs + and − and on the Early German Arithmeticians," *Messenger of Mathematics* **51**, 1–148 (1921–1922).

H. Grassman, *Hermann Grassmanns Gesammelte Mathematische und Physicalische Werke*, F. Engel, Editor, B. G. Teubner, Leipzig (1894); includes: "Die Wissenschaft der extensiven Grösse oder die Ausdehnungslehre, ein neue mathematische Disciplin," Leipzig (1844); also "Geometrische Analyse" (1846). For Inner and Outer Products see: (1844), pp. 77–102, and (1846), pp. 345–351.

R. P. Graves, *The Life of Sir William Rowan Hamilton*, 3 volumes, Dublin (1882–1891).

Sir W. R. Hamilton, *Elements of Quaternions*, First Edition, London (1865), Second Edition, London (1899).

Sir W. R. Hamilton, see also *Elements of Quaternions* (1866) and *The Mathematical Papers* (1967).

T. L. Hankins, *Sir William Rowan Hamilton*, The Johns Hopkins University Press, Baltimore and London (1980).

T. L. Heath, *Diophantus of Alexandria: A Study in the History of Greek Algebra*, 2nd Edition, Cambridge (1910).

J. van Heijenoort, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, Harvard University Press, Cambridge, MA (1967).

V. F. Hopper, *Medieval Number Symbolism: Its Sources, Meaning, and Influence on Thought and Expression*, Columbia University Press, New York (1938).

E. W. Hyde, "Calculus of Direction and Position," *American Journal of Mathematics* **6**, 1–13 (1884).

G. Ifrah, *From One to Zero: A Universal History of Numbers*, Viking Penguin Inc., New York (1985); English translation of "Histoire Universelle des Chiffres."

K. E. Iverson, "Notation as a Tool of Thought," 1979 Turing Award Paper, *Communications of the ACM* **23**, No. 8, 444–465 (August 1980).

W. S. Jevons, *The Principles of Science: A Treatise on Logic and Scientific Method*, First Edition (1874); 2nd Edition (1877). Reprinted by Dover Publications Inc., New York, with introduction by Ernest Nagel (1958).

L. C. Karpinski, *The History of Arithmetic*, Rand McNally & Co., Chicago and New York (1925).

G. Lamé, *Leçons sur la Théorie Mathématique de l'Elasticité des Corpes Solides*, 1st Edition, Bachelier, Paris (1852).

H. Lass, *Vector and Tensor Analysis*, McGraw-Hill Book Company, Inc., New York (1950).

C. I. Lewis, *A Survey of Symbolic Logic*, University of California (1918). Reprinted by Dover Publications Inc., New York (1960).

C. I. Lewis and C. H. Langford, *Symbolic Logic*, Princeton University, Princeton, NJ (1932). Extract: "History of Symbolic Logic" included in *The World of Mathematics*, Vol. 3, J. R. Newman, Editor, Simon and Schuster, New York (1956), pp. 1859–1877.

A. Macfarlane, "Lectures on Ten British Mathematicians of the Nineteenth Century," *Mathematical Monographs*, No. 17, John Wiley & Sons, Inc., New York (1916).

E. Mach, *The Science of Mechanics: A Critical and Historical Account of Its Development*, First German Edition (1883); English translation, Chicago (1902). Reprinted by The Open Court Publishing Company, La Salle, IL (1960).

D. MacHale, *George Boole: His Life and Work*, Boole Press, Dublin (1985).

The Mathematical Papers of Sir William Rowan Hamilton, Vol. 3—*Algebra*, H. Halberstram and R. E. Ingram, Editors, Cunningham Memoir No. 15, Cambridge University Press, New York (1967).

D. B. McIntyre, "Experience with Direct Definition One-Liners in Writing APL Applications," *An APL Users Meeting, Proceedings*, I. P. Sharp Associates, Ltd., Toronto (1978), pp. 281–297.

D. B. McIntyre, "Introduction to the Study of Data Matrices," *Models of Geologic Processes—an Introduction to Mathematical Geology*, P. Fenner, Editor, American Geological Institute, Washington, DC (1969).

O. Neugebauer, "The Exact Sciences in Antiquity," *Acta Historica Scientiarum Naturalium et Medicinalium*, Vol. 9, Copen-

hagen (1951), and Princeton University, Princeton, NJ, and Oxford (1951). Reprinted by Harper Torchbooks, Harper, New York (1962), and by Dover Publications Inc., New York.

R. P. Polivka and S. Pakin, *APL: The Language and Its Usage*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1975); see p. 192 for Boolean product A <. < B.

G. Prasad, *Some Great Mathematicians of the Nineteenth Century, Their Lives and Their Work*, 3 volumes, The Benares Mathematical Society, India (1934).

J. E. Quibell (with notes by W. M. F. P), *Hierakonpolis, Part I, Plates of Discoveries in 1898*, Egyptian Research Account, 4th Memoir, Bernard Quaritch, London (1900), Plate 26B.

G. Sarton, "The First Explanation of Decimal Fractions and Measures (1585), Together with a History of the Decimal Idea and a Facsimile (No. xvii) of Stevin's *Disme*," *Isis* **23**, 153–244 (1935).

G. Sarton, "Simon Stevin of Bruges (1548–1620)," *Isis* **21**, 241–303 (1934).

G. Sarton, *The Study of the History of Mathematics*, Cambridge, MA (1936).

J. F. Scott, *A History of Mathematics from Antiquity to the Beginning of the Nineteenth Century*, Taylor and Francis, London (1960).

D. E. Smith, "Algebra of 4000 Years Ago," *Scripta Mathematica* **4**, 111–125 (1936).

D. Smith, *Interface: Calculus and the Computer*, First Edition, Chapter 12; Second Edition, Saunders (1984), p. 63–75; Instructor's Manual, pp. 23–25.

D. E. Smith, "The Law of Exponents in the Works of the 16th Century," in *Napier Tercentenary Memorial Volume*, C. G. Knott, Editor, London (1915), pp. 81–91.

D. E. Smith, *A Source Book in Mathematics*, McGraw-Hill Book Company, Inc., New York and London (1929), especially pp. 217–228.

D. E. Smith, *The Teaching of Elementary Mathematics*, Macmillan Co., London (1901).

D. E. Smith and J. Ginsburg, "From Numbers to Numerals and from Numerals to Computation," from *Numbers and Numerals*. Reprinted in *The World of Mathematics*, Vol. 1, J. R. Newman, Editor, Simon and Schuster, New York (1956), p. 442–464.

D. E. Smith and L. C. Karpinski, *The Hindu-Arabic Numerals*, Boston and London (1911).

S. Stevin, *The Principal Works of Simon Stevin*, 5 volumes, Volume 1, *General Introduction*, *Mechanics*, E. J. Dijksterhuis, Editor, C. V. Swets and Zeitlinger, Amsterdam (1955).

D. J. Struik, *A Concise History of Mathematics*, 2 volumes, Dover Publications Inc., New York (1948).

F. J. Swetz, *Capitalism and Arithmetic: The New Math of the 15th Century*, The Open Court Publishing Company, La Salle, IL (1987).

F. Thureau-Dangin, "Sketch of a History of the Sexagesimal System," *Osiris* **7**, 95–141 (1939).

H. A. Thurston, *The Number-System*, Interscience Publishers, Inc., New York (1964).

J. Venn, "Boole's Logical System," *Mind* **1**, 479–491 (1876).

J. Venn, "George Boole, 1815–1864," *Dictionary of National Biography*.

M. Ward and C. E. Hardgrove, *Modern Elementary Mathematics*, Addison-Wesley Publishing Co., Reading, MA (1964).

W. Whewell, *History of the Inductive Sciences from the Earliest to the Present Time*, 3 volumes, Vol. 2, *History of Mechanics*, Second Edition, John W. Parker, London (1847).

E. T. Whittaker, "The Sequence of Ideas in the Discovery of Quaternions," Quaternion Centenary Celebration, 8th November, 1943, Royal Irish Academy, *Proceedings* **A50**, pp. 93–98 (1945).

R. L. Wilder, *Evolution of Mathematical Concepts: an Elementary Study*, John Wiley & Sons, Inc., New York (1968). Reprinted by The Open University, Milton Keynes (1978).

E. B. Wilson, *Vector Analysis: A Text-Book for the Use of Students Founded Upon the Lectures of J. Willard Gibbs*, Charles Scribner's Sons, New York (1909). Reprinted by Dover Publications Inc., New York (1960).

Accepted for publication August 5, 1991.

Donald B. McIntyre *Luachmhor, Church Road, Kinfauns, Perth PH2 7LD, Scotland, U.K.* Donald McIntyre was educated in Scotland, receiving B.Sc., Ph.D., and D.Sc. degrees from Edinburgh University where he was a member of the faculty from 1948–1954. He did postdoctoral research at the University of Neuchâtel, the University of California at Berkeley, and the Dominion Observatory, Canada. From 1954 until 1989, he was Professor of Geology at Pomona College. In addition to teaching geology, he has been active for 30 years in computing, obtaining one of the first IBM System/360 computers in 1965, the second of IBM's 5100 series, and the second of IBM's 4300 series in May of 1979. He has lectured around the world for Sigma Xi, the American Association of Petroleum Geologists, the British Museum, the Geological Society of America, the ACM as a Distinguished Lecturer, the State Seismological Bureau in Beijing, the University of Nanjing in China, and numerous universities in the United States, Canada, Britain, and Europe. In 1969 he gave the Matthew Vassar Lecture on the subject of APL at Vassar College. In 1971, he was a consultant with the APL group at the IBM Scientific Center in Philadelphia under A. Falkoff and K. Iverson. He has received a Fulbright Award, a John Simon Guggenheim Memorial Fellowship, and in 1985 was named California College and University Professor of the Year by the Council for the Advancement of Support of Education. Donald McIntyre is now retired in his native Scotland, but is still active in the use and promotion of APL. He is an Honorary Fellow at the Universities of Edinburgh and St. Andrews.

Reprint Order No. G321-5454.

A personal view of APL

by K. E. Iverson

This essay portrays a personal view of the development of several influential dialects of APL: APL2 and J. The discussion traces the evolution of the treatment of arrays, functions, and operators, as well as function definition, grammar, terminology, and spelling.

It is now 35 years since Professor Howard Aiken instituted a computer science program at Harvard, a program that he called *Automatic Data Processing*. It is almost that long since I began to develop, for use in writing and teaching in that program, the programming language that has come to be known as APL.

Although I have consulted original papers and compared my recollections with those of colleagues, this remains a personal essay that traces the development of my own thinking about notation. In particular, my citation of the work of others does not imply that they agree with my present interpretation of their contributions. In speaking of design decisions I use the word *we* to refer to the small group associated with the early implementation, a group that included Adin Falkoff, Larry Breed, and Dick Lathwell, and is identified in "The Design of APL"[1] and "The Evolution of APL."[2] These papers contain full treatments of various aspects of the development of APL that are given scant attention here.

Because my formal education was in mathematics, the fundamental notions in APL have been drawn largely from mathematics. In particular, the notions of arrays, functions, and operators were adopted at the outset, as illustrated by the following excerpt from *A Programming Language*.[3]

> An operation (such as summation) which is applied to all components of a vector is called reduction. . . . Thus, $+/x$ is the sum, $\times/x$ is the product, and $\vee/x$ is the logical sum of the components of a vector x.

The phrase $+/x$ alone illustrates the three aspects: a *function* +, an *operator* / (so named from the term used by Heaviside[4] for an entity that applies to a function to produce a related derived function), and an *array* x.

The present discussion is organized by topic, tracing the evolution of the treatments of arrays, functions, and operators; as well as that of other matters such as function definition, grammar, terminology, and spelling (that is, the representation of primitives).

As stated at the outset, the initial motive for developing APL was to provide a tool for writing and teaching. Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects.

When I retired from paid employment, I turned my attention back to this matter and soon concluded that the essential tool required was a dialect of APL that:

- Is available as "shareware," and is inexpensive enough to be acquired by students as well as by schools
- Can be printed on standard printers
- Runs on a wide variety of computers
- Provides the simplicity and generality of the latest thinking in APL

The result has been J, first reported in Reference 5.

Work began in the summer of 1989 when I first discussed my desires with Arthur Whitney. He proposed the use of C for implementation, and produced (on one page and in one afternoon) a working fragment that provided only one function (+), one operator (/), one-letter names, and arrays limited to ranks 0 and 1, but did provide for boxed arrays and for the use of the copula for assigning names to any entity.

I showed this fragment to others in the hope of interesting someone competent in both C and APL to take up the work, and soon recruited Roger Hui, who was attracted in part by the unusual style of C programming used by Arthur, a style that made heavy use of preprocessing facilities to permit writing further C in a distinctly APL style.

Roger and I then began a collaboration on the design and implementation of a dialect of APL (later named J by Roger), first deciding to roughly follow "A Dictionary of APL"[6] and to impose no requirement of compatibility with any existing dialect. We were assisted by suggestions from many sources, particularly in the design of the spelling scheme (E. B. Iverson and A. T. Whitney) and in the treatment of cells, items, and formatting (A. T. Whitney, based on his work on SHARP/HP[7] and on the dialect A reported at the APL89 conference in New York).

E. E. McDonnell of Reuters provided C programs for the mathematical functions (which apply to complex numbers as well as to real), D. L. Orth of IBM ported the system to the IBM RISC System/6000* in time for the APL90 conference, and L. J. Dickey of the University of Waterloo provided assistance in porting the system to a number of other computers.

The features of J that distinguish it from most other APL dialects include:

1. A spelling scheme that uses ASCII characters in one- or two-letter words
2. Convenient international use, provided by facilities for alternative spellings for the *national use* characters of ASCII, and by facilities to produce the error messages in any desired language
3. Emphasis on *major cells* or *items*; for example, reduction (`f/`) applies `f` between items, and application of `f` between cells of lesser rank is obtained by using the rank operator
4. The function argument to scan (`\`) is, like all functions, ambivalent. Scan applies the monadic case of the function rather than the dyadic. Thus, the traditional sum scan is given by `+/\a` rather than by `+\a`, and `<\a` boxes the partitions provided by the scan.
5. A number of other partitioning adverbs are provided, including *suffix scan* (`\.`), windows of width *k* (as in `k f\a`), and *oblique* (`/.`).
6. Use of the hook and fork (discussed later) and various new operators together with the use of the copula to assign names to functions. These facilities permit the extensive use of *tacit* programming in which the arguments of a function are not explicitly referred to in its definition, a form of programming that requires no reparsing of the function on execution, and therefore provides some of the efficiency of compilation. (See Reference 8.)
7. An immediate and highly readable display of the definition of a function `f` obtained by simply entering `f`

Significant use of J in teaching will, of course, require the development of textual material using it. Three steps have been taken toward this goal:

1. The dictionary of J includes 45 frames of tutorial material (suitable for slides) that are brief treatments in J of topics from a dozen different areas.
2. At the urging of L. B. Moore of I. P. Sharp Associates, I prepared for distribution at APL89 a booklet called *Tangible Math*, designed for independent study of elementary mathematics. It was based on the use of Sharp[9] shareware for the IBM PC, and required no reference to an APL manual. I have since produced a J version of *Tangible Math*.[10]
3. At a four-hour hands-on workshop for teachers of mathematics organized by Anthony Camacho of I-APL[11] and funded by the British APL Association, Anthony and I used *Tangible Math* to expose the participants to the advantages of ex-

ecutable mathematical notation. The teachers left with a copy of J and with enough experience to continue the use of J on their own. Such workshops could be used to bring teachers to a point where they could develop their own treatments of isolated topics, and eventually of complete subjects, on their own.

In the three decades of APL development, many different ideas have been proposed and explored, and many have been abandoned. Those that survived have done so through incorporation in one or more implementations that define the many dialects of APL.

These dialects fall into several families, two of which have been particularly influential. I refer to them by the names of their most recent exemplars—APL2[12] on the one hand, and J on the other—and sketch the development of these families in a later section.

In the remainder of the essay I largely confine my remarks to those dialects that have influenced, and been influenced by, my own thinking. This emphasis is intended not to denigrate the dialects not mentioned, but to keep the discussion focused and to leave their exposition to others more conversant with them.

Although my motive for producing a new dialect was for use in teaching, this dialect has led to much greater emphasis on a style of programming called *functional* by Backus,[13] and defined in J as *tacit* programming (because arguments are not referred to explicitly). These matters are addressed in the section on tacit programming.

Terminology

Although terminology was not among the matters given serious attention at the outset, it will be helpful to adopt some of the later terminology immediately. Because of our common mathematical background, we initially chose mathematical terms. For example, the sentence

```
b←(+\a)-.×a←2 3 5 7
```

illustrates certain parts of speech, for which we adopted the mathematical terms shown on the left as follows:

Functions or operators	+ × −	Verbs
Constant (vector)	2 3 5 7	Noun (list)
Variables	*a b*	Pronouns
Operator	\	Adverb
Operator	.	Conjunction
	( )	Punctuation
	←	Copula

I now prefer terms drawn from natural language, as illustrated by the terms shown on the right. Not only are they familiar to a broader audience, but they clarify the purposes of the parts of speech and of certain relations among them:

1. A verb specifies an "action" upon a noun or nouns.
2. An adverb applies to a verb to produce a related verb; thus +\ is the verb "partial sums."
3. A conjunction applies to two verbs, in the manner of the copulative conjunction *and* in the phrase "run and hide."
4. A name such as *a* or *b* behaves like a pronoun, serving as a surrogate for any referent linked to it by a copula. The mathematical term *variable* applied to a name x in the identity $(x+1)\times(x+3)$ equals x^2+4x+3 serves to emphasize that the relation holds for *any* value of x, but the term is often inappropriate for pronouns used in programming.
5. Although numeric lists and tables are commonly used to represent the vectors and matrices of mathematics, the terms *list* and *table* are much broader and simpler, and suggest the essential notions better than do the mathematical terms.
6. To avoid ambiguity due to the two uses of the term *operator* in mathematics (for both a function and a Heaviside operator) I usually use only the terms *adverb* and *conjunction*, but continue to use either *function* or *verb*, *list* or *vector*, and *table* or *matrix*, as seems appropriate.

Spelling

In natural languages the many words used are commonly represented (or *spelled*) in an *alphabet* of a small number of characters. In programming languages the words or *primitives* of the languages (such as *sin* and =:) are commonly represented by an expanded alphabet that includes a number of graphic symbols such as + and =.

When we came to implement APL, the alphabet then widely available on computers was extremely limited, and we decided to exploit a feature of our

company's newly-developed Selectric* typewriter, whose changeable typing element allowed us to design our own alphabet of 88 characters. By limiting the English alphabet to one case (majuscules), and by using the backspace key to produce *composite* characters, we were able to design a spelling scheme that used only one-character words for primitives.

Moreover, the spelling scheme was quite mnemonic in an international sense, relying on the appearance of the symbols rather than on names of the functions in any national language. Thus the phrase $k \uparrow x$ *takes* k elements from x, and $\downarrow$ denotes *drop*.

Because the use of the APL alphabet was relatively limited, it was not included in the standard ASCII alphabet now widely adopted. As a consequence, it was not available on most printers, and the printing and publication of APL material became onerous. Nevertheless, in spite of some experiments with "reserved words" in the manner of other programming languages, the original APL alphabet has remained the standard for APL systems.

The set of graphics in ASCII is much richer than the meager set available when the APL alphabet was designed, and it can be used in spelling schemes for APL primitives that still avoid the adoption of reserved words. Such a scheme using variable-length words was presented in Reference 6, and received limited use for communicating APL programs using standard printers, but was never adopted in any commercial implementation. A much simpler scheme using words of one or two letters was adopted in J, in a manner that largely retains, and sometimes enhances, the international mnemonic character of APL words.

In a natural language such as English, the process of word formation is clearly distinguished from parsing. In particular, word formation is static, the rhematic rules applying to an entire text quite independently of the meanings or grammatical classes of the words produced. Parsing, on the other hand, is dynamic, and proceeds according to the grammatical classes of phrases as they evolve. This is reflected in the use of such terms as *noun phrase* and *verb phrase*.

In programming languages this distinction is commonly blurred by combining word formation and parsing in a single process characterized as "syntax." In J, the word formation and parsing are distinct. In its implementations, each process is *table-driven*; the parsing table being presented explicitly in the dictionary of J, and the rhematic rules being discussed only informally.

It is interesting to note that the words of early APL included "composite characters" represented by two elements of the underlying alphabet; these were mechanically superposed, whereas in J they appear side-by-side.

I largely confine my remarks to those dialects that have influenced my own thinking.

Functions

Functions were first adopted in the forms found in elementary mathematics, having one argument (as in $|b|$ and $-b$) or two (as in $a+b$ and $a-b$). In particular, each had an explicit result, so that functions could be articulated to form sentences, as in $|a-b| \div (a+b)$.

In mathematics, the symbol $-$ is used to denote both the *dyadic* function *subtraction* (as in a−b) and the *monadic* function *negation* (as in −b). This ambivalent use of symbols was exploited systematically (as in $\div$ for both *division* and *reciprocal*, and * for both *power* and *exponential*) to provide mnemonic links between related functions, and to economize on symbols.

The same motivations led us to adopt E. E. McDonnell's proposal to treat the monadic *trigonometric* (or *circular*) functions and related *hyperbolic* and *pythagorean* functions as a single family of dyadic functions, denoted by a circle. Thus *sine y* and *cosine y* are denoted by $1 \circ y$ and $2 \circ y$, the numeric left argument being chosen so that its parity (even or odd) agrees with the parity of the function denoted, and so that a negative integer denotes the function inverse to that denoted by the corresponding positive integer. This scheme was a matter of following (with rather less justification) the impor-

tant mathematical notion of treating the monadic functions *square*, *cube*, *square root*, etc. as special cases of the single dyadic power function.

When the language was formalized and linearized in APL\360,[14] anomalies such as x^y for power, xy for product, $|y|$ for magnitude, and M^i_j for indexing were replaced by `x*y` and `x×y` and `|y` and `M[i;j]`. At the same time, function definition was formalized, using *headers* of the form `Z←X F Y` and `Z←F Y` to indicate the definition of a dyadic or a monadic function. This form of header permitted the definition of functions having no explicit result (as in `X F Y`), and so-called *niladic* functions (as in `Z←F` and `F`) having no explicit arguments. These forms were adopted for their supposed convenience, but this adoption introduced functions whose articulation in sentences was limited.

Box and enclose have made it convenient to pass any number of parameters as explicit arguments.

In most later dialects such niladic and resultless functions were also adopted as primitives. In J they have been debarred completely, to avoid the problem of articulation, to avoid complications in the application of adverbs and conjunctions to them, and to avoid the following problem with the copula: if `g` is a niladic function that yields the noun `n`, and if `f←g`, then is `f` a niladic function equivalent to `g`, or is it the noun `n`?

In conventional mathematical notation, an expression such as *f(x,y,z)* can be interpreted either as a function of three arguments, or as a function of one argument, that is, of the vector formed by the catenation of *x*, *y*, and *z*. Therefore the limitation of APL functions to at most two formal arguments does not limit the number of scalar arguments to which a function may apply.

Difficulties with nonscalar arguments first arose in indexing, and the forms such as `A[I;J;K]` and `A[I;;K]` that were adopted to deal with it introduced a "nonlocality" into the language: a phrase within brackets had to be treated as a whole rather than as the application of a sequence of functions whose results could each be assigned a name or otherwise treated as a normal result. Moreover, an index expression for an array `A` could not be written without knowing the rank of `A`.

The introduction of a function to produce an *atomic representation* of a noun (known as *enclose* in NARS[15,16] and APL2, as *box* in SAX[17] and J, and discussed in the section on atomic representations) makes it possible to box arguments of any rank and assemble them into a single argument for any function. In particular, it makes possible the use of such a boxed array as the argument to an indexing function, adopted in SAX and J and called *from*.

As may be seen,[18] the function *rotate* was initially defined so that the right argument specified the amount of rotation. The roles of the arguments were later reversed to accord with a general mnemonic scheme in which a left argument `a` together with a dyadic function `f` (denoted in J by `a&f`) would produce a "meaningful" monadic function. Exceptions were, of course, made for established functions such as *divided by*. The scheme retains some mnemonic value, although the commute adverb (`~`) provided in J and in SAX makes either order convenient to use. For example, `5 %~ 3` would be read as *5 into 3*.

In APL\360 it was impossible to define a new function within a program. This was rectified in APLSV[19] by defining a *canonical representation* of a function (a matrix `M` whose first row was a header, and whose succeeding rows were the sentences of the definition); a *fix* function `□FX` such that `□FX M` yielded the name of the function as an explicit result, and established the function as a side effect; and an inverse function `□CR`, which when applied to the name of a function produced its canonical representation as an explicit result. The ability to define ambivalent functions was added in a University of Massachusetts system,[20] and was soon widely adopted.

The function `□FX` established a function only as a side effect, but the scheme has been adapted to J by providing a *conjunction* (`:`) such that `m : d` produces an unnamed function that may be applied directly, as in `x m : dy`, or may be assigned a name, as in `f=.m : d`. See the section on name assignment.

Following an idea that Larry Breed picked up at a lecture by the late Professor A. Perlis of Yale, we

adopted a scheme of *dynamic localization* in which names localized in a function definition were known to further functions invoked within it.

This decision made it possible to pass any number of parameters to subordinate functions, and therefore circumvented the limitation of at most two explicit arguments, but it did lead to a sometimes confusing profusion of names localized at various levels. The introduction of atomic representation (box and enclose) has made it convenient to pass any number of parameters as explicit arguments; in J this has been exploited to allow a return to a simpler localization scheme in which any name is either strictly local or strictly global.

Arrays

Perhaps because of the influence of a course in tensor analysis taken as an undergraduate, I adopted the notion that every function argument is an array, and that arrays may be classified by their *ranks*; a scalar is rank *0*, a vector rank *1*, a matrix rank *2*, and so on.

The application of arithmetic (or *scalar*) function such as + and × also followed tensor analysis; in particular the *scalar extension*, which allowed two arguments to differ in rank if one were a scalar. In defining other functions (such as reshape and rotate), we attempted to make the behavior on higher-rank arrays as systematic as possible, but failed to find a satisfying uniform scheme. Such a uniform scheme (based on the notion of *cells*) is defined in "A Dictionary of APL,"[6] and adopted in SAX and in J.

A *rank-k cell* of `A` is a subarray of `A` along `k` contiguous final axes. For example, if:

```
   A
abcd
efgh
ijkl

mnop
qrst
uvwx
```

then the list `abcd` is a 1-cell of `A`, the table from `m` to `x` is a 2-cell of `A`, the atom `g` is a 0-cell of `A`, and `A` itself is a 3-cell of `A`.

Each primitive function has *intrinsic* ranks, and applies to arrays as a collection of cells of the appropriate rank. For example, matrix inverse has rank 2, and applies to an array of shape `5 4 3` as a collection of five `4` by `3` matrices to produce a result of shape `5 3 4`, a collection of five `3` by `4` inverses of the `4` by `3` cells.

Moreover, the rank conjunction (denoted in J by ") produces a function of specified rank. For example, the intrinsic rank of ravel is unbounded and (using the shape `2 3 4` array `A` shown above):

```
   ,A
abcdefghijklmnopqrstuvwx

   , "2 A
abcdefghijkl
mnopqrstuvwx
```

Further discussion of cells and rank may be found in the section on tacit programming, and in Reference 21.

The central idea behind the use of cells and a rank operator was suggested to me at the 1982 APL conference in Heidelberg by Arthur Whitney. In particular, Arthur showed that a reduction along any particular axis (`+/[I]A`) could be neatly handled by a rank operator, as in `+/"I A`. By further adopting the idea that every primitive possessed intrinsic ranks (monadic, left, and right) I was able, in Reference 6, to greatly simplify the definition of primitives: each function need be defined only for cells having the intrinsic ranks, and the extension to higher-rank arguments is uniform for all functions.

Adverbs and conjunctions

Even after tasting the fruits of generalizing the Σ notation of mathematics to the form `f/` that permitted the use of functions other than addition, it took some time before I recognized the advantages of a corresponding generalization of the *inner* or *matrix* product to allow the use of functions other than addition and multiplication. Moreover, I thought primarily of the derived functions provided by these generalizations, and neither examined the nature of the slash itself nor recognized that it behaved like a Heaviside operator.

However, when we came to linearize the notation in the implementation of APL\360, the linearization of the inner product (which had been written as one function on top of the other) forced the adoption of a symbol for the conjunction (as in `M +. × N`). This

focused attention on the adverbs and conjunctions themselves, leading to a recognition of their role and to the adoption of the term *operators* to refer to them.

In reviewing the syntax of operators we were disturbed to realize that the slash used for reduction applied to the (function) argument to its *left*, and even considered the possibility of reversing the order to agree with the behavior of monadic functions. However, Adin Falkoff soon espoused the advantages of the established scheme, pointing out that the adoption of a "long left scope" for operators would allow the writing of phrases such as `+.×/` to denote the function "inner product reduction," which might be applied to a rank-3 array.

We also realized that the use of the slash to denote *compression* (as in `1 0 1 0 1/'abcde'` to yield `'ace'`) seemed to imply that the slash was ambiguous, sometimes denoting an operator, and sometimes a function. This view was adopted in NARS and in the precursor to APL2. Alternatively, adverbs and conjunctions could be assumed to apply to both nouns and verbs, giving different classes of derived verbs in the different cases. In this view, compression was not a dyadic function denoted by the slash, but was rather the derived function resulting from the application of the adverb / to a noun.

The application of adverbs and conjunctions to nouns was adopted in SHARP,[22] SHARP/HP, SAX, and J, but was resisted in other dialects, in spite of the fact that the phrase `⌽[3]` for applying reversal on axis 3 furnished an example of such usage in early APL, and in spite of the implied use of nouns in Heaviside's notation D^2 `f` for the second derivative of `f`.

In calculus, the expression f+g is used to denote the sum of functions *f* and *g*, that is, (f+g) x is defined as (f x)+(g x). The utility of such constructs as f+g and f×g was clear, and I realized that they could be handled by operators corresponding to the functions + and ×. What appeared to be needed was an adverb that would apply to a function to produce a *conjunction*. However, I was reluctant to complicate the grammar by introducing results other than functions from adverbs, and I began by suggesting, in Reference 23, a limited solution using composite symbols such as + overstruck by an overbar.

Somewhat later I discussed this matter with Arthur Whitney, and he quickly suggested an operator that we modified slightly and presented as the *til* operator in Reference 24, using the definition `x (f til g) y` is `(g y) f x`. The *fork* discussed in the section on grammar and order of execution now provides a more convenient solution, using expressions such as `f+g` and `f×g`.

In mathematics, the notions of inner product and outer product are used in rather limited areas. In APL systems, operators provide generalizations of them that not only broaden their uses, but make them more readily comprehensible to nonmathematicians. Much the same is true of "duals" in mathematics, but because the generalization of APL is not so widely known or used, it merits some attention here.

The need for parentheses will be reduced by executing compound statements from right to left.

It is useful to view almost any task as performed in three phases: preparation, the main task, and undoing the preparation. In programming terms this would appear as `inversep main p argument`. In other words, the main function is performed *under* the preparation `p`.

In J the *under* conjunction is denoted by `&.` and is defined as follows:

```
   m&.p y is inversep m p y
 x m&.p y is inversep (p x) m (p y)
```

For example, since `^.` denotes the natural logarithm in J, the expression `a +&.^. b` yields the product of `a` and `b`. The *under* conjunction is commonly used with the function *open* (whose inverse is *box*) discussed in the section on atomic representations.

Name assignment

In mathematics, the symbol = is used to denote both a relation and the copula in name assignment

(as in "let x=3"). In APL, the arrow was first used for the copula in Reference 18, and has been used in all dialects until the adoption of =. and =: in J.[21]

The use of the copula was initially restricted to nouns, and names were assigned to user-defined functions by a different mechanism in which the name of the function was incorporated in the representation to which the function □*FX* was applied, as discussed in the previous section on functions. The use of the copula for this purpose was proposed in Reference 23, implemented in SHARP/HP, and later adopted in Dyalog[25] and in J. These implementations provided for adverbs and conjunctions in the same manner. However, this use of the copula has not been adopted in other implementations, perhaps because the representations used for functions make its adoption difficult.

Indirect assignment was first proposed in Reference 26, and is implemented in J and defined in Reference 21. Two copulas are used in J, one for local assignment (=.), and one for global (=:) assignment.

Grammar and order of execution

Grammatical rules determine the order of execution of a sentence, that is, the order in which the phrases are interpreted. In Reference 3, the use of parentheses was adopted as in mathematics, together with the rule (Reference 3, page 8) that "The need for parentheses will be reduced by assuming that compound statements are, except for intervening parentheses, executed from right to left."

In particular, this rule implies that there is no hierarchy among functions (such as the rules in mathematics that power is executed before multiplication before addition). Long familiarity with this hierarchy occasioned a few lapses in my book,[3] but the new rule was strictly adopted in the APL\360 implementation. APL\360 also *introduced* a hierarchy, giving operators precedence over functions.

The result was a simple grammar, complicated only by the bracket-semicolon notation used for indexing. This was later complicated by the adoption, in most systems, of the *statement separator* (denoted by a diamond). The utility of the statement separator was later vitiated in some systems (including SHARP, SAX, and J) by the adoption of dyadic functions *lev* and *dex*, which yielded their left and right arguments, respectively.

The grammatical rules left certain phrases (such as a sequence of nouns) invalid. In NARS and in APL2 meanings were assigned to a sequence of nouns: if *a* and *b* are the nouns "hold" and "on," then the phrase `a b` yields the two-element list of enclosed vectors. The adoption of such "strands" led to a modification of the grammatical rules based upon left and right "binding strengths" assigned to various parts of speech, as discussed in References 27 and 28. In particular these rules required that the phrase `2 3 5[1]` be replaced by `(2 3 5)[1]`.

Other changes in grammar were adopted in J: the bracket-semicolon indexing was replaced by a normal dyadic verb *from*; and any isolated sequence of verbs was assigned a meaning based upon the *hook* and *fork*, first proposed in Reference 29 and briefly explained next. The result is a strict grammar in which each phrase for execution is chosen from the first four elements of the execution stack, and eligibility for execution is determined by comparison with a 14 by 4 parsing table as shown in Reference 21.

Because the hook and fork (as well as several other previously invalid phrases) play a significant role in the *tacit* programming discussed in a later section, they are further elaborated here. Briefly, if

```
mean=.+/%#
```

then

```
mean x
```

is equivalent to

```
(+/x)%(#x)
```

The dyadic case is defined analogously. If

```
diffsq=. +*-
```

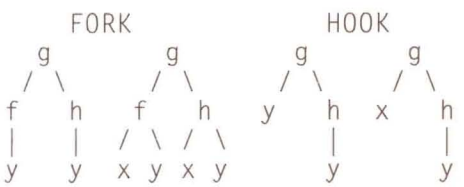
then

```
a diffsq b
```

is

```
(a+b)*(a-b)
```

The hook and the fork may be expressed graphically as follows:



Two further points should be noted:

1. A longer train of verbs will resolve into a sequence of forks and hooks. For example, `taut=. <: =<+.=` is equivalent to two forks, as in `taut=. <: = (< +. =)`, and expresses the tautology that *less than or equal* (`<:`) *equals* (`=`) *less than* (`<`) *or* (`+.`) *equal* (`=`).
2. In the expression `(+/ % #) 2 3 4 5` to produce the mean of the list `2 3 4 5`, the parentheses are clearly essential, since `+/ % # 2 3 4 5` would yield 0.25, the sum of the reciprocal of the number of items. However, it must be emphasized that the parentheses perform their normal function of grouping, and are not needed to explicitly produce forks, as may be seen from the earlier examples.

Atomic representations

It is commonplace that complex constructs may be conveniently represented by arrays of simpler constructs: a word by a list of letters, a sentence by a list of words, a complex number by a list of two real numbers, and the parameter of a rotation function by a table of numbers, and so on.

However, it is much more convenient to use *atomic* representations, which have rank 0 and are therefore convenient to combine into, and select from, arrays. For example, the representation `3j4` used for a complex number in APL systems is an *atom* or *scalar*.

In Reference 30, Trenchard More proposed a representation scheme in which an *enclose* function applied to an array produced a scalar representation of the argument. This notion was adopted or adapted in a number of APL systems, beginning with NARS, and soon followed by APL2.

A somewhat simpler scheme was adopted in SHARP in 1982, was presented in "A Dictionary of APL"[6] in 1987, and later adopted in SAX and J: a function called *box* (and denoted by `<`) applied to any noun produces an atomic representation of the noun that can be "decoded" by the inverse function *open* (denoted by `>`) to yield the original argument.

A desire for similar convenience in handling collections of functions led Bernecky and others to propose (in References 31 and 32) the notion of *function arrays*. These have been implemented as *gerunds* in J by adopting atomic representations for functions.

Implementations

Because of a healthy emphasis on standardization, many distinct implementations differed slightly, if at all, in the language features implemented. For example, the IBM publication APLSV User's Manual[19] written originally for APLSV applied equally to VS APL and the IBM 5100 computer.

Despite the present emphasis on the evolution of the language itself, certain implementations merit mention:

1. The IBM 5100 mentioned above is noteworthy as one of the early desktop computers, and as an implementation based on an emulator of the IBM System/360* and a read-only memory copy of APLSV.
2. The I-APL implementation provided the first shareware version of APL, aimed at making APL widely available in schools.

Implementations representing the two main lines of development mentioned in the introduction are now discussed briefly. The first is the *nested array system* NARS conceived and implemented by Bob Smith of STSC and incorporating ideas due to Trenchard More[30] and J. A. Brown (Doctoral thesis, University of Syracuse). In addition to the *enclose* and related facilities that provide the nested arrays themselves, this implementation greatly expanded the applicability of operators. In the APL2 implementation, Brown has followed this same line of development of nested arrays.

Somewhat after the advent of NARS, the SHARP APL system was extended to provide *boxed* elements in arrays, as reported in Reference 22. New operators (such as the *rank*) were also added, but their utility was severely limited by the fact that operators were not (as in NARS) extended to apply to user-defined functions and derived functions. In the succeeding SAX and J implementations such constraints have been removed.

Tacit programming

A *tacit* definition is one in which no explicit mention is made of the arguments of the function being defined. For example:

```
   sum=. +/
   mean=. sum % #
   listmean=. mean"1

   [a=. i. 5
0 1 2 3 4

   sum a
10

   mean a
2

   [table=. i. 3 5
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14

   mean table
5 6 7 8 9

   listmean table
2 7 12
```

By contrast, definition in most APL dialects makes explicit mention of the argument(s):

```
      □FX 2 7⍴'Z←SUM X Z←+/X'
SUM
```

Tacit programming offers several advantages, including the following:

1. It is concise.
2. It allows significant formal manipulation of definitions.
3. It greatly simplifies the introduction of programming into any topic.

Since the phrase +/ produces a function, the potential for tacit programming existed in the earliest APL; but the restrictions on the copula prevented assignment of a name to the definition, and therefore prohibited tacit programming.

In any case, the paucity of operators and the restrictions that permitted their application to (a subclass of) primitive functions only, made serious use of tacit programming impossible. In later dialects these restrictions have been removed, and the number of operators has been increased.

I now provide a few examples of tacit programming in J, first listing the main facilities to be exploited. The reader may wish to compare such facilities in J with similar facilities defined by Backus[13] and by Curry.[33] For example, Curry's combinators W (elementary duplicator) and C (commutator) are both represented by the adverb ~ in J, according to the following examples:

```
/:~b    is   b/:b (that is, a sort of b)
a %~b   is   b%a  (that is, a into b)
```

The facilities to be used in the examples include the *hook*, *fork*, and ~ already defined, as well as the following which, although defined in terms of specific verbs, apply generally. It may be necessary to consult Reference 21 for the meanings of certain verbs, such as *: (square), %: (square root), and ^. (log). Five examples follow.

```
1.    2 &^ y is 2^y          (Called currying)
2.    ^ & 2 y is y^2         (Called currying)
3.    -&^.y   is -^.y        Composition
4. x -&^. y is (^.x)-(^.y)   Composition
5. x -@^ y    is - x^y       Atop
```

Some examples from statistics are shown next.

```
sum=. +/
mean=. sum % #
norm=. - mean
std=.%: & sum & *: & norm
```

Entry of a function alone causes a display of its definition, a display that can be captured and manipulated as a straightforward boxed array. Thus:

```
   std
+-----------------+-+----+
|+---------+-+---+|&|norm|
|| +-+-+---+|&|*:||  |    |
|| |%:|&|sum|| |  ||  |    |
|| +-+-+---+| |  ||  |    |
|+---------+-+---+|  |    |
+-----------------+-+----+
```

In function tables, the f outer product of APL is in J the dyadic case of f/. For example:

```
   [a=. b=. i. 5
0 1 2 3 4

   a +/ b
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

   a*/b
0 0 0  0  0
0 1 2  3  4
0 2 4  6  8
0 3 6  9 12
0 4 8 12 16

   a!/b
1 1 1 1 1
0 1 2 3 4
0 0 1 3 6
0 0 0 1 4
0 0 0 0 1
```

Such a table can be made easier to interpret by displaying it with appended arguments, using the following tacit definitions:

```
   over=.({.,.@;}.)&":@,
   by=. (,~"_1 ' '&;&,.)~
   a by b over a !/ b
+-+---------+
| |0 1 2 3 4|
+-+---------+
|0|1 1 1 1 1|
|1|0 1 2 3 4|
|2|0 0 1 3 6|
|3|0 0 0 1 4|
|4|0 0 0 0 1|
+-+---------+
```

Adverbs may be defined tacitly in a number of ways, as follows:

```
   sum \ a
0 1 3 6 10

   scan=. /\
   + scan a
0 1 3 6 10

   - scan a
0 _1 1 _2 2
```

```
   table=. /(['by']'over')\
   2 3 5 *table 1 2 3 4 5
+-+-------------+
| |1  2  3  4  5|
+-+-------------+
|2|2  4  6  8 10|
|3|3  6  9 12 15|
|5|5 10 15 20 25|
+-+-------------+

   a <table b
+-+---------+
| |0 1 2 3 4|
+-+---------+
|0|0 1 1 1 1|
|1|0 0 1 1 1|
|2|0 0 0 1 1|
|3|0 0 0 0 1|
|4|0 0 0 0 0|
+-+---------+
```

* Trademark or registered trademark of International Business Machines Corporation.

Cited references and note

1. A. D. Falkoff and K. E. Iverson, "The Design of APL," *IBM Journal of Research and Development* **17**, No. 4, 324–334 (1973).
2. A. D. Falkoff and K. E. Iverson, "The Evolution of APL," *ACM SIGPLAN Notices* **13**, No. 8, 47–57 (1978).
3. K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York (1962), p. 16.
4. See the 1971 Chelsea edition of Heaviside's *Electromagnetic Theory* and the article by P. Nahin in the June 1990 issue of *Scientific American*.
5. R. K. W. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney, "APL/?," *APL90 Conference Proceedings*, *APL Quote Quad* **20**, No. 4, ACM, New York (1990).
6. K. E. Iverson, "A Dictionary of APL," *APL87 Conference Proceedings*, *APL Quote Quad* **18**, No. 1, 202–211, ACM, New York (1987).
7. R. Hodgkinson, "APL Procedures," *APL86 Conference Proceedings*, *APL Quote Quad* **16**, No. 4, ACM, New York (1986).
8. R. K. W. Hui, K. E. Iverson, and E. E. McDonnell, "Tacit Programming," *APL91 Conference Proceedings*, *APL Quote Quad* **21**, No. 4, ACM, New York (1991).
9. P. C. Berry, *Sharp APL Reference Manual*, I. P. Sharp Associates, Toronto, Canada (1979).
10. K. E. Iverson, *Tangible Math*, Iverson Software Inc., Toronto, Canada (1990).
11. A. Camacho, "I-APL Status Report," *Vector: The Journal of the British APL Association* **4**, No. 3, 8–9 (1988).
12. *APL2 Programming: System Services Reference*, SH20-9218, IBM Corporation (1988); available through IBM branch offices.
13. J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* **21**, No. 8, 613–641, (1978).

14. A. D. Falkoff and K. E. Iverson, *APL\360 User's Manual*, IBM Corporation (1966).
15. R. Smith, "Nested Arrays, Operators, and Functions," *APL81 Conference Proceedings*, *APL Quote Quad* **12**, No. 1, ACM, New York (1981).
16. C. M. Cheney, *Nested Arrays Reference Manual*, STSC Inc., Rockville, MD (1981).
17. *SAX Reference*, 0982 8809 E1, I. P. Sharp Associates, Toronto, Canada (1986).
18. K. E. Iverson, "The Description of Finite Sequential Processes," *Proceedings of a Conference on Information Theory*, C. Cherry and W. Jackson, Editors, Imperial College, London (August 1960).
19. *APLSV User's Manual*, GC26-3847-3, IBM Corporation (1973).
20. C. Weidmann, *APLUM Reference Manual*, University of Massachusetts (1975).
21. K. E. Iverson, *The ISI Dictionary of J*, Iverson Software Inc., Toronto, Canada (1991).
22. R. Bernecky and K. E. Iverson, "Operators and Enclosed Arrays," *APL User's Meeting*, I. P. Sharp Associates, Toronto, Canada (1980).
23. K. E. Iverson, *Operators and Functions*, Research Report 7091, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1978).
24. A. T. Whitney and K. E. Iverson, "Practical Uses of a Model of APL," *APL82 Conference Proceedings*, *APL Quote Quad* **13**, No. 1, ACM, New York (1982).
25. *Dyalog APL Reference Manual*, Dyadic Systems Ltd., Alton, Hants, England (1982).
26. K. E. Iverson, "APL Syntax and Semantics," *APL83 Conference Proceedings*, *APL Quote Quad* **13**, No. 3, 223–231, ACM, New York (1983).
27. J. P. Benkard, "Valence and Precedence in APL Extensions," in *APL83 Conference Proceedings*, *APL Quote Quad,* **13**, No. 3, ACM, New York (1983).
28. J. D. Bunda and J. A. Gerth, "APL Two by Two—Syntax Analysis by Pairwise Reduction," *APL84 Conference Proceedings*, *APL Quote Quad* **14**, No. 4, ACM, New York (1984).
29. K. E. Iverson and E. E. McDonnell, "Phrasal Forms," *APL89 Conference Proceedings*, *APL Quote Quad* **19**, No. 4, ACM, New York (1989).
30. T. More, Jr., "Axioms and Theorems for a Theory of Arrays," *IBM Journal of Research and Development* **17**, No. 2, 135–157 (1973).
31. R. Bernecky, "Function Arrays," *APL84 Conference Proceedings*, *APL Quote Quad* **14**, No. 4, ACM, New York (1984).
32. J. A. Brown, "Function Assignment and Arrays of Functions," *APL84 Conference Proceedings*, *APL Quote Quad* **14**, No. 4, ACM, New York (1984).
33. H. B. Curry and R. Feys, *Combinatory Logic*, Vol. 1, North Holland Publishers, Amsterdam, Netherlands (1968).

Accepted for publication June 25, 1991.

Kenneth E. Iverson *70 Erskine Avenue, No. 405, Toronto, Ontario M4P 1Y2, Canada.* Dr. Iverson received a B.A. in mathematics and physics from Queen's University, Kingston, Canada in 1950, an M.A. in mathematics in 1951, and a Ph.D. in applied mathematics from Harvard University. He was an assistant professor at Harvard from 1955 to 1960. From 1960 to 1980 he was employed by IBM Corporation's Research Division where he became an IBM Fellow in 1970. After leaving IBM in 1980, Dr. Iverson was employed by I. P. Sharp Associates until 1987. He has received many honors, in addition to becoming an IBM Fellow, including the AFIPS Harry Goode Award in 1975, the ACM Turing Award in 1979, and the IEEE Computer Pioneer Award in 1982. He is a member of the National Academy of Engineering in the United States. Currently he is working on J and the use of J in teaching.

Reprint Order No. G321-5455.

Books

Exploring Requirements: Quality Before Design, Donald C. Gause and Gerald M. Weinberg, Dorset House Publishing, New York, 1989. 291 pp. (ISBN 0-932633-13-7).

This book comes as a breath of fresh air at a time when system designers can begin to feel stifled by the formal methodologies and computer-aided design technologies that are the focus of attention. Gause and Weinberg provide us with the assurance that it is still the human mind, not the machine, that is the system designer's most important tool.

The subject of the book is the requirements process, the earliest part of the development cycle in which designers attempt to discover what is desired. Part I, "Negotiating a Common Understanding," begins by telling us why methodologies are not enough. Gause and Weinberg are true masters of the anecdotal example, a technique they often use to underscore their points.

Most significant system problems found in test and operational use (where the cost of fixing errors is greatest) can be traced back to ambiguity in the requirements. Early attacking of ambiguity lowers development costs in the end. Unfortunately, system designers are so anxious to get going that they often ignore or minimally perform the requirements exploration phase. If every designer assigned to a new project were to heed the advice in this book, it would surely prolong the requirements analysis, but the overall development process would be made more efficient. Inefficiencies caused by requirements ambiguity is at the heart of this book's message, and there are many sources for ambiguity. The authors' style is unique in that they do not simply enumerate those sources, but allow the reader to experience the ambiguities firsthand. In fact, much of the book gives the reader the sense of attending a live lecture rather than reading a book.

Part II addresses every system designer's biggest problem—getting started. The authors' techniques serve to slow down the beginning of a project, allowing the mind to do its work of grasping a better understanding of the problem. A chapter on "Getting the Right People Involved" provides the practical means to accomplish today's market-driven approach of involving the user throughout the system definition, design, and development phases. The message Gause and Weinberg give is clear, which is that the practice of definition, design, and development and system understanding are the same.

"Exploring Possibilities" is the subject of Part III. These possibilities include providing more how-to techniques, conducting idea-generation meetings, using right-brain methods, and selecting a project name so as to have a theme or rallying point toward which to work. An excellent discussion is made of facilitation and facilitators, a concept that the authors expect to reach maturity in the '90s. Part IV, "Clarifying Expectations," brings us still farther along the path of eliminating requirements ambiguity by addressing functions, attributes, constraints, preferences, and expectations. A number of subtle yet key points are made. For example, "only the strength of the client's fears or desires determines which is a constraint and which is a preference." Such thought-provoking assertions are often brought home by an enjoyable anecdote to give the reader an innate understanding of the key point. The how-to techniques offered in the book are highly usable and by following them the reader should achieve positive results.

The final part of the book, "Greatly Improving on the Odds of Success," addresses ambiguity metrics, technical reviews, measuring satisfaction, test cases, study of existing products, and making agreements.

The authors conclude with a little philosophy on ending the requirements exploration phase. In each chapter they include a section on helpful hints, variations, and a useful summary that captures the ideas by using those tried and true editorial stalwarts: why, when, how, and who.

The pace of the book is very comfortable. It ebbs and flows such that every once in a while a burst of excitement comes with a new idea, followed by explanations that allow the reader to ride the wave. The numerous diagrams, done with intentional informality, keep an interesting and personal tone.

The principles discussed can most definitely be applied by a committed team and management, and they should contribute to the success of a project. It is up to creative individuals to find ways to apply the principles within their own environments, through the inspiration of the authors. Gause and Weinberg foresee the '90s as a time in which use of systems with higher complexity will grow. They envision more reliance on computer-aided design technology, and customers with higher expectations. Gause and Weinberg provide practical methodologies and techniques that allow system designers to bring their project-development practices to a level that meets expected system, technology, and environmental complexity.

Donna H. Rhodes
IBM Federal Sector Division
Owego
New York

Knowledge Engineering, Dimitris N. Chorafas, Van Nostrand Reinhold, New York, 1990. 380 pp. (ISBN 0-442-23969-6).

This book is aimed at computer professionals who want to upgrade their skills, and stay competitive in understanding the field of knowledge engineering and the methodology of knowledge acquisition. The author envisions a bright future for knowledge engineering and artificial intelligence (AI), in general.

Despite the author's optimistic forecast for the field, he does not minimize the many obstacles between "here" and "there." The discussions fairly balance the difficulties and advantages of specific approaches. There are a liberal number of schematic illustrations. A minimal amount of technical jargon is used for a book dealing with this type of subject and, for this reason, it should be easy to read for a diverse audience.

The book represents a comprehensive survey of the field of knowledge engineering. It provides an overview of what is currently going on for those who have not been directly involved with AI. For people who already are involved, it may suggest other projects where the same skills are useful. A part of the author's purpose seems to be to attract new people to the field. He suggests that the need for such experts will grow 25 percent per year during this decade. Viewed in this light, the book may well open new vistas of opportunity. Although it provides little insight into the detail required for the actual implementation of programs, this may perhaps be presumed by his target audience of computer professionals.

Included in the book is an appendix that outlines a training program for knowledge engineers. However, this is not a textbook in the sense of a book that might form the basis for a curriculum. The book is a comprehensive survey and, as such, should appeal to a number of people.

The specific examples cited tend to be discussed from the perspective of business applications, rather than research or scientific contributions. Although the bibliography contains 200 references, these are not cited in the text; the few citations that do occur in the text do not appear in the bibliography. The index also is not as helpful as one might desire. In short, the book does not serve as a scholarly treatise or as a reference source.

The author is a consultant who has written 65 books. In many ways, this book reminds me, in expanded form, of some of the reports I have received from consultants. Such reports tend to be comprehensive, easy to read, well organized, and noncontroversial, though they do not expand the field. While the author may be correct in predicting the future success of the field of AI, eventual success may not directly depend on many of the current methods and theories, but may require as yet unforeseen breakthroughs.

The book goes into enough depth to make its significant points. It is easy to read, is up to date, and

covers most topics of current interest. I would recommend it to anyone contemplating knowledge engineering as a possible vocation, and to anyone who wants an overview of what this might entail.

W. D. Hagamen, M.D.
Professor of Cell Biology and Anatomy
Cornell University Medical College
New York

VSAM: A Comprehensive Guide, Constantine Kaniklidis, Van Nostrand Reinhold, New York, 1990. 440 pp. (ISBN 0-442-24641-2).

If you are interested in acquiring a good overall knowledge of how VSAM works, this is definitely the book for you. Constantine Kaniklidis has done an excellent job of consolidating vast amounts of available information into one easily comprehensible book. He also provides the reader with many good practical suggestions for using VSAM effectively, with occasional editorial comments expressing his likes and dislikes of the access method.

The book begins by presenting the reader with a short evolutionary synopsis of VSAM from its inception in 1973 up to the present. It then proceeds to describe, in detail, the functional components of VSAM, including catalog and data set structure and operation, and the use of IDCAMS in defining, loading, and listing catalogs and data sets. With the functional components having been laid out, several chapters follow that present various options for data set security, backup, and recovery, with the final chapter being devoted to VSAM optimization, including such topics as control interval size calculation, free space distribution, buffer allocation, and data set sharing. Throughout the entire book, each topic is accompanied by examples that help to clarify the information presented.

Having spent many of my years at IBM working on the VSAM access method, I think that the reader will find this book very informative. While the information included in the book appears to be technically accurate, it is difficult to evaluate its factual contents down to the "bits and bytes" level, due to the volume and detail of information presented and the ever-changing nature of the subject. It is therefore my recommendation that the reader consult the appropriate IBM publication for verification whenever a high degree of detail is required.

Gale A. Burt
IBM Storage Systems Products Division
San Jose
California

Note—The books reviewed are those the Editor thinks might be of interest to our readers. The reviews express the opinions of the reviewers.

Contents of Volume 30, 1991

Erratum

The paper "Transaction Security System" by D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens that appeared in the *IBM Systems Journal*, Vol. 30, No. 2, pages 206 through 229, omitted reference to the source of the signature verification algorithm, pen design, pen data acquisition and signal processing design, and signature recognition reliability data used in this system. The authors regret the omission and now cite the following as a key reference to this work:

T. K. Worthington, T. J. Chainer, J. D. Williford, and S. C. Gundersen, "IBM Dynamic Signature Verification," Computer Security: The Practical Issues in a Troubled World, *Proceedings of the Third IFIP International Conference on Computer Security*, Dublin, Ireland (1985), pp. 129–154.

The *IBM Systems Journal* is abstracted by *Chemical Abstracts, Computer Abstracts, Computer & Control Abstracts, Computer & Information Systems, Computer Literature Index, Data Processing Digest, Ekspress Informatsiia, Electrical & Electronics Abstracts, The Engineering Index, Information Science Abstracts, Mathematical Reviews, New Literature on Automation (Netherlands), Operations Research/Management Science, Referativnyi Zhurnal,* and *Science Citation Index*. Reviews appear in *Computing Reviews*. Reproductions of the *IBM Systems Journal* by years are available on microfiche and positive and negative microfilm from University Microfilms, 300 N. Zeeb Road, Ann Arbor, Michigan 48106 U.S.A. An electronic version of the *IBM Systems Journal* is available as part of a comprehensive database of periodicals distributed by the Computer Library Division of Ziff Communications Company, One Park Avenue, New York, New York 10016 U.S.A.

IBM Systems Journal APL Volume 30, Number 4, pp. 413-598 • Index,1991

G321-0105-00