APL Since 1978

ROGER K.W. HUI, Dyalog Limited, Canada
MORTEN J. KROMBERG, Dyalog Limited, United Kingdom

Shepherd: Guy L. Steele Jr., Oracle Labs, USA

The Evolution of APL, the HOPL I paper by Falkoff and Iverson on APL, recounted the fundamental design principles which shaped the implementation of the APL language in 1966, and the early uses and other influences which shaped its first decade of enhancements.

In the 40 years that have elapsed since HOPL I, several dozen APL implementations have come and gone. In the first decade or two, interpreters were typically born and buried along with the hardware or operating system that they were created for. More recently, the use of C as an implementation language provided APL interpreters with greater longevity and portability.

APL started its life on IBM mainframes which were time-shared by multiple users. As the demand for computing resources grew and costs dropped, APL first moved *in-house* to mainframes, then to *mini-* and *micro*-computers. Today, APL runs on PCs and tablets, Apples and Raspberry Pis, smartphones and watches.

The operating systems, and the software application platforms that APL runs on, have evolved beyond recognition. Tools like database systems have taken over many of the tasks that were initially implemented in APL or provided by the APL system, and new capabilities like parallel hardware have also changed the focus of design and implementation efforts through the years.

The first wave of significant language enhancements occurred shortly after HOPL I, resulting in so-called second-generation APL systems. The most important feature of the second generation is the addition of general arrays—in which any item of an array can be another array—and a number of new functions and operators aligned with, if not always motivated by, the new data structures.

The majority of implementations followed IBM's path with APL2 "floating" arrays; others aligned themselves with SHARP APL and "grounded" arrays. While the APL2 style of APL interpreters came to dominate the mainstream of the APL community, two new cousins of APL descended from the SHARP APL family tree: J (created by Iverson and Hui) and k (created by Arthur Whitney).

We attempt to follow a reasonable number of threads through the last 40 years, to identify the most important factors that have shaped the evolution of APL. We will discuss the details of what we believe are the most significant language features that made it through the occasionally unnatural selection imposed by the loss of habitats that disappeared with hardware, software platforms, and business models.

The history of APL now spans six decades. It is still the case, as Falkoff and Iverson remarked at the end of the HOPL I paper, that:

> Although this is not the place to discuss the future, it should be remarked that the evolution of APL is far from finished.

Authors' addresses: Roger K.W. Hui, Dyalog Limited, Canada, roger@dyalog.com; Morten J. Kromberg, Dyalog Limited, Minchens Court, Minchens Lane, Bramley, Hampshire, RG26 5BH, United Kingdom, morten@dyalog.com.

2475-1421/2020/6-ART69
https://doi.org/10.1145/3386319

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: APL, programming languages, array programming, functional programming, higher-order functions, executable mathematical notation

ACM Reference Format:
Roger K.W. Hui and Morten J. Kromberg. 2020. APL Since 1978. *Proc. ACM Program. Lang.* 4, HOPL, Article 69 (June 2020), 108 pages. https://doi.org/10.1145/3386319

ACKNOWLEDGMENTS

We thank Bob Bernecky, Jim Brown, Adám Brudzewsky, Gitte Christensen, Jay Foad, Simon Garland, Brent Hailpern, Aaron Hsu, Michael Hughes, Eric Iverson, Mike Jenkins, Jon McGrew, Roger Moore, Ray Polivka, Henry Rich, John Scholes, Bob Smith, Fiona Smith, Richard Smith, Geoff Streeter, Roy Sykes, Joey Tuttle, and Arthur Whitney for their comments and suggestions.

We thank Guy Steele and the other anonymous referees for their comments and suggestions. We are indebted to Guy Steele for "shepherding" the paper.

The writing of the paper was supported by a grant from First Derivatives plc and its Kx Division.

> Once launched on a topic it's very easy to forget to mention the contributions of others, and although I have a very good memory, it is, in the words of one of my colleagues, very short, so I feel I must begin with acknowledgments, …
>
> — K.E. Iverson, *The Evolution of APL* [Falkoff and Iverson 1978]

CONTENTS

0. Historical Perspective 4
   0.1 Platforms 4
   0.2 Applications and Interfaces 6
   0.3 Evolutionary Branches 8
   0.4 System Integration 11
   0.5 APL Around the World 14
   0.6 APL Since 1978 14
About the APL Examples 20
1. User Definitions 21
   1.1 Control Structures 21
   1.2 Direct Functions 22
   1.3 Trains 23
   1.4 Naming 25
2. Functions 26
   2.1 *Tally* ≢ 26
   2.2 *Index-Of* ⍳ *et al.* 27
   2.3 *Interval Index* ⍸ 29
   2.4 *Index* [] 29
   2.5 *Grade* ⍋ 31
   2.6 Should It Be a Primitive? 32
3. Operators 33
   3.1 *Rank* ⍤ 34
   3.2 *Power* ⍣ 39
   3.3 *Key* ⌸ 40
   3.4 *Stencil* ⌺ 41
   3.5 *Under* ⍢ and *Obverse* ⍥ 43
   3.6 *Each* ¨ 44
   3.7 *At* @ 45
   3.8 Trains Encore 46
4. Arrays 47
   4.1 Simple Homogeneous Arrays 47
   4.2 Nested and Boxed Arrays 47
   4.3 Objects 48
   4.4 Futures and Isolates 50
   4.5 Sparse Arrays 52
   4.6 Infinite Arrays 53
5. Syntax 55
   5.1 Parsing an APL Expression 55
   5.2 Parentheses 57
   5.3 Function and Operator Valence 57
   5.4 Left *v* Right Argument 58
   5.5 Numerals 58
6. Symbols 59
   6.1 Overview 59
   6.2 Vailing and Countervailing Pressures 59
   6.3 Composing Symbols 61
   6.4 Exercises 63
   6.5 Names 63
7. User Interfaces 65
   7.1 Textual User Interfaces 65
   7.2 Microsoft Win32 66
   7.3 .Net and Other Object Frameworks 66
   7.4 Data Binding 67
   7.5 APL as a Service 67
8. Compilers 68
   8.1 Lack of Adoption of Early Compilers 68
   8.2 Potential Benefits of Compilers 69
   8.3 Parallel Compilers 69
   8.4 Special Purpose Compilers 70
9. Implementation 72
   9.1 Array Representations 72
   9.2 Small-Range Data 73
   9.3 Idioms (Special Code) 73
   9.4 Magic Functions 74
   9.5 `assert` 75
10. Currents and Eddies 76
   10.1 Terminology 76
   10.2 What's in a Name? 77
   10.3 Backward Compatibility 78
   10.4 Index Origin 78
   10.5 APL Characters 79
11. Code 81
   11.1 Random Numbers 81
   11.2 Quicksort 82
   11.3 Permutations 83
   11.4 Symmetries of the Square 85
   11.5 Total Resistance 86
   11.6 Ackermann Function 87
   11.7 Ken Iverson's Favorite APL Expression? 88

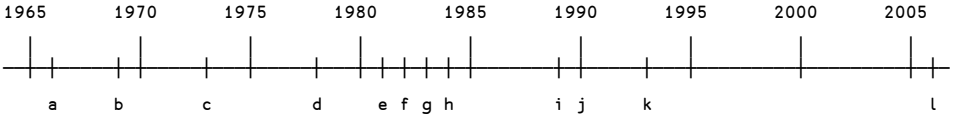
Appendices 89
A. *APL\360* Acknowledgments 89
B. Summary of Notation 90
C. A Parser Model 91
   C.1 Description 91
   C.2 `parse` the Function 94
   C.3 Auxiliary Data and Functions 95
   C.4 Tests 96
D. *Dramatis Personæ* 98
E. Alternative Histories 99
F. Photo Credits 99

References 99

0 HISTORICAL PERSPECTIVE

In 1954, Howard Aiken of Harvard University instituted a graduate program in computer science called "automatic data processing", the first such program in the world [Iverson 1954a; Iverson 1991a; Brooks Jr. 1999, p.141]. He directed Kenneth E. Iverson, a recent Ph.D. in applied mathematics under his supervision, to implement a course in this program. Iverson needed notation for describing and analyzing various topics in the course, and was "appalled" that conventional mathematical notation failed to fill the need. He proceeded to develop notation that was suitable [Iverson and McIntyre 2008]. In 1960, Iverson left Harvard to join IBM Research where he continued to work on the notation, soon known informally among colleagues as "Iverson notation". The notation eventually became APL, as described in the Acknowledgments in *APL\360 User's Manual* [Falkoff and Iverson 1968] and reproduced in Appendix A.

The HOPL I paper *The Evolution of APL* [Falkoff and Iverson 1978] described the state of APL in 1978. From thence we pick up the story. APL since 1978.



a	1966	*APL\360* available at IBM Research	[Falkoff and Iverson 1967]
b	1969	IPSA (I.P. Sharp Associates) and STSC (Scientific Time Sharing Corp.) launch APL PLUS time-sharing service	[Iverson 1978a; IPSA 1979]
c	1973	IBM releases APLSV	[Falkoff and Iverson 1973a]
d	1978	*Operators and Functions* published	[Iverson 1978b]
e	1981	STSC releases NARS	[Cheney 1981]
f	1982	STSC releases APL*PLUS/PC	[STSC 1983]
g	1983	Dyalog APL version 1.0	[Dyalog 2008a]
h	1984	IBM releases APL2	[Brown 1984]
i	1989	Arthur Whitney creates A at Morgan Stanley	[Whitney 1989]
j	1990	J version 1.0	[Hui et al. 1990]
k	1993	Arthur Whitney creates k at Kx Systems	[Whitney 1993]
l	2006	Bob Smith launches NARS2000	[Smith 2020]

The main focus of this paper is the evolution of the APL language. However, we begin with a discussion of the impact that changes to hardware and software platforms have had on APL vendors and users in the 40 years since *The Evolution of APL*. Unlike the timeline above, the discussion is topical rather than chronological: platforms (§0.1), applications and interfaces (§0.2), branches of the APL evolutionary tree (§0.3), system intergration (§0.4), and APL around the world (§0.5), concluding with a transition to the history of APL since 1978 (§0.6).

0.1 Platforms

As our story begins, APL was most widely used by large corporations sharing a small number of mainframes in Toronto, managed by IPSA (I.P. Sharp Associates), and in Bethesda, Maryland, managed by STSC (Scientific Time Sharing Corp.). IBM's APL systems were located in Kingston, NY, Sterling Forest, NY, San Jose, CA, among other locations; they served approximately 30,000 users, almost all internal to IBM. Starting in 1976, as the cost of hardware came down, and the demand for analytical computing went up, many IPSA/STSC clients followed the example of IBM

APL users in running APL in-house on mainframes [IPSA 1978; Iverson 1982]. APL on the mainframe provided a "personal computing" environment to subject-matter experts at all levels of organizations including top management, decades before this became commonplace. Users had direct access to corporate data through very simple APIs, and there were truly state-of-the-art tools for creating UIs on 24×80 (or 25×80) monitors. APL on the mainframe was a uniquely productive environment, providing both personal computing and a platform for corporate multi-user systems in the same package.

It was not obvious to users of APL that the technological revolution that began with minicomputers and quickly moved on to microcomputers provided any real benefits. APL had often been selected as a tool precisely because it did not require much of an understanding of computer and network technology, allowing users to build applications that encapsulated solutions expressed in an executable mathematical notation. As a result, APL users and vendors were generally slow to understand the implications of—and respond to—the revolution that was taking place around them. (There was one notable exception: Micro Computer Machines of Canada was first to market with the MCM/70 APL microcomputer. The attaché version marked the first use of a personal computer on an airline flight in August 1973, on its way to a European demonstration tour, starting at APL Congress 73 in Copenhagen [Stachniak 2011, p. 66-68].)

0.1.1 Minicomputers

Minicomputers caused a brief burst of activity during which every manufacturer of minicomputers (various flavors of DEC (Digital Equipment Corp.) and Hewlett-Packard machines, Data General, Prime, Philips, and Honeywell, to name a few) all needed an APL interpreter in order to take business off the mainframe. However, the world quickly moved on to microcomputers, where the operating systems were: MS-DOS, dominating the corporate market; UNIX, for "technical" applications (which included a lot of financial analytics done in APL); and a little later Microsoft Windows and the Apple Macintosh.

0.1.2 MS-DOS

On the MS-DOS platform, STSC quickly emerged as the clear winner with the APL*PLUS/PC system, which was only a first-generation APL interpreter but had excellent interfaces to PC peripherals and was used as a base for many "shrink-wrapped" APL products, including Statgraphics. APL*PLUS/PC was followed by a 32-bit, second-generation family of products known as APL*PLUS/II, which subsequently became the APL+Win product that is on the market today.

Even for the most successful players, the revenues from PC license sales were much smaller than the old time-sharing revenues, and even STSC suffered greatly and went through cycles of downsizing and acquisitions. IPSA moved a handful of clients to SHARP APL for UNIX [Steinbrook 1986] running on Sun workstations, but never managed to get a real foothold in the new markets.

IBM was not hit as hard since most of their revenue came from hardware and operating system licenses, which were significantly more robust. However, IBM reduced its investment in APL, one reason being the Systems Application Architecture initiative from 1987, which demanded standardization on APIs and programming languages. Products were required to be available on all IBM hardware platforms in order to be part of SAA. This included the AS/400 range (previously System/38), a database machine without good floating-point support, and no APL.

0.1.3 New APL Vendors

In addition to the efforts of the big three (IBM, IPSA, and STSC) and the minicomputer manufacturers, a number of new APL interpreters were started up in this period targeting workstations and microcomputers, often in collaboration with a company producing an up-market workstation.

MicroAPL created APL.68000, which later became APLX and became the leading APL system for the Macintosh.

Many of these systems contained original ideas for language features and system interfaces, especially the systems that were implemented for special hardware architectures, such as APL for the Ampere and Analogic machines, but the authors have insufficient insight into these stories to report on them.

In 1982, an APL consultancy in the UK called Dyadic Systems Ltd. decided that a good way to protect their consulting business would be to write their own second-generation APL system. The chip manufacturer Zilog thought that an APL interpreter would be a good way to showcase the Z8000, their new 16-bit replacement for the 8-bit Z80 microprocessor. The two partnered in a project to produce an APL for UNIX using NARS as the blueprint and C as the implementation language [Cheney 1981; Dyalog 2008a]. The result was Dyalog APL. ("Dyalog" is an amalgam of "Dyadic" and "Zilog".)

Selling APL under UNIX on the Zilog platform in 1983 failed miserably, but fortunately Dyadic Systems was acquired by Lynwood, a terminal manufacturer, who wanted to use a graphics package written in APL to showcase a new line of terminals. That idea didn't fly either. After a long period of hardship where the key developers John Scholes and Geoff Streeter sometimes worked without pay, Dyalog APL became a commercially viable product around 1990, with support for MS-DOS and Microsoft Windows in addition to UNIX, and started to generate enough revenue to support a development team. (The key people also bought the company back from Lynwood in March 1990.) Since that time, Dyalog has steadily acquired market share and is currently investing heavily in APL language design and implementation.

0.1.4 Luck

Inevitably, luck has played an important role in deciding the fates of many APL systems. As an extreme example of this, Dyadic Systems was gifted a significant market share in Europe, essentially because it was lagging behind STSC in the race to produce an interpreter targeting the new Windows NT platform. STSC was using a 32-bit extension called Win32s that Microsoft provided to allow applications to continue to run on the more popular 16-bit Windows until everyone upgraded to Windows NT. At the same time, a significant fraction of large corporations in Europe decided to mandate the use of IBM OS/2 for "end user computing". There were almost no applications available, so the result was that a lot of users ended up living in the OS/2 "Windows Compatibility Box", which had limited support for Windows applications. Dyalog APL could run there, but Win32s-based applications could not.

One can discuss the extent to which luck played a role in the next story: IBM's PC XT/370 architecture, which runs IBM mainframe applications on a set of boards inserted into an IBM PC XT, led some players (including IPSA) in the APL market astray, by diverting limited development resources and delaying market entry. To a problem-oriented APL user-implementer, the XT/370 was obviously the system of the future. Unsurprisingly (at least in hindsight), IBM decided against selling mainframe replacements at a fraction of the price of a real one.

0.2 Applications and Interfaces

As mentioned in the previous section, APL on the mainframe was a highly productive environment which allowed engineers, actuaries, and managers to write models and crunch numbers. APL users had efficient access to data in all the various file formats that the mainframe had to offer, state-of-the-art systems for producing 24×80 "fullscreen" user interfaces, and budding business graphics systems. Operators monitored the machine 24 hours a day, took backups—and restored them on demand.

0.2.1 The Rug is Pulled

While the arrival of personal computers provided most people with the new-found freedom offered by spreadsheets and other personal productivity tools, many APL users already had everything they needed—including freedom—on the mainframe. To them, the PC sometimes meant no access—or at least less reliable, slower access to the corporate data that they used to have at their fingertips. There were no operators taking backups or keeping the machine well-oiled.

Worse, the technical skill required to write applications suddenly increased dramatically. You needed to know about networks, how to deal with the poor reliability of shared files on a LAN—or how to construct client/server components to achieve the security, performance, and reliability that you needed. User interfaces, which were so simple on the mainframe, became an absolute nightmare as an endless procession of new GUI frameworks and UI "standards" appeared and then faded away.

For software engineers who trained for years to do refactoring, this was an exciting time. But for someone who wanted to solve problems with a computer and had been able to do it easily in APL on the mainframe, it was a disaster. The highly productive environment in which smart people who had other primary skills than programming–such as bond trading or chemical engineering—could build reliable multi-user applications, was replaced by a jumble of unreliable, rapidly evolving APIs created for software engineers.

APL vendors had always taken pride in wrapping APIs in array-oriented mechanisms which would feel natural to APL users and hide the technical details as far as possible. For many APL vendors, already struggling as the value of the market collapsed after years of milk and honey on the mainframe, the challenge of trying to wrap the steady stream of APIs to create user interfaces or access databases proved too much of a challenge.

At the same time, many of the problems that had been solved with hand-made solutions became available as shrink-wrapped solutions, further reducing the value of the market. IBM and IPSA stopped marketing APL in any meaningful sense in this period, although the products continued to exist. STSC underwent a number of acquisitions and was weakened each time. All the minicomputers disappeared at the end of the 1980's or early 1990's. The number of APL vendors dropped precipitously, and APL largely disappeared from the curricula of higher educational institutions.

0.2.2 The Turn of the Tide

A handful of APL products, like APLX from MicroAPL, Dyalog APL, and APL+Win from APL2000 (successor of STSC), managed to provide wrappers for GUI APIs, high performance database connectivity, and TCP wrappers—that allowed APL teams to be competitive. Using these and other tools, a number of companies using APL were able to adapt to the new world and provide subject-matter experts with tools that allowed them to continue to use APL, without the bulk of the APL users having to learn too much engineering.

Even in the hostile environment of the DotCom bubble, where corporations had almost unlimited funds to standardize on a single programming language (typically C++ or Java), APL survived in places where the value of being able to involve subject-matter experts directly in the writing of code was sufficiently high.

Since the 1990s, the environment has been more friendly. From an organizational perspective, the software community has become much more aware of the value of involving subject-matter experts, and APL can now be explained as an extreme branch of agile software development, where the users write code too. The use of multiple programming languages within an organization is now acceptable; the idea that a single language could be the "best fit" for solving every type of problem has faded. Array orientation, efficient use of memory, and functional programming have all become hot topics, and APL is slowly being recognized again as a respectable tool.

An important development is that computing platforms have evolved to make it easy to connect together modules written in different languages. Microsoft's .NET Framework showed the way with self-describing Common Language Runtime (CLR) modules specifically designed to be language agnostic, and the Java ecosystem has followed suit. The Internet, and now cloud computing, have accelerated the drive towards simple, modular solutions and away from "big frameworks".

Today, the recommended architecture for new applications designed for the cloud is microservices, hosted in containers, based on minimalistic frameworks, small operating systems (UNIX is back in the form of Linux), using inverted, in-memory databases, connected together via sockets that are used to transmit self-describing data in the form of JSON or XML messages. The execution environment is—from a user perspective—indistinguishable from the mainframes of 1978. There are people oiling the machine again, and even taking backups.

We have almost come full circle, and find ourselves in a relatively simple environment, well suited to writing solutions in a functional array language like APL. REST (Representational State Transfer) is a software architecture for creating web services [Fielding 2000]. Self-describing RESTful APIs provide access to the data we need, and any amount of computing power is available on demand.

We are now ready to examine how the APL language itself has evolved over the last 40 years.

0.3 Evolutionary Branches

In the beginning, there was IBM's *APL*\360 [Falkoff and Iverson 1967; Pakin 1972]. The APL PLUS system started as a licensed copy of *APL*\360 and further developed by two of the three original APL implementers, Roger Moore (IPSA) and Larry Breed (STSC) [Iverson 1978a]. When STSC built its own data center in Bethesda, Maryland, in the early 1970s, the two companies pursued different strategies, and SHARP APL and APL*PLUS became separate products.

Features added subsequently enhanced (hence the name) the ability to build commercial systems that would consume the machine cycles and the international networking services that generated time-sharing revenues. These included FORTRAN-style report formatting and a file system which could store APL arrays for efficient use in multi-user systems. In contrast, IBM preferred to write interfaces to other components also provided by IBM on the mainframes that they sold to large corporations.

In 1978, the differences between APL systems then available were still quite small, but increasing rapidly. Initially, APL arrays had to be homogeneous, rectangular collections of either characters or numbers (§4.1). In the 1970s, the idea of general arrays, where any item of an array can be another array, started to form (§4.2). The prevailing design of IBM's APL2 was based on work by Jim Brown and the array theory developed by Trenchard More; primitive scalar functions apply "pervasively" to items of arrays, not only at the outer level but also into the nested structures [Brown 1971, 1984].

0.3.1 Floating *v* Grounded

Iverson was uncomfortable with many of these ideas. In addition to pervasion, APL2 specified a function *enclose* such that the enclose of a scalar number or character returns the scalar unchanged ("floating" arrays). Iverson felt strongly that *enclose*, or "box", should produce a different data type which was outside the domain of mathematical functions, and always increase the level of nesting by 1 ("grounded" arrays). Iverson proposed a set of new operators that would explicitly apply functions to items within a nested structure [Iverson 1978b]. (See also §4.2.)

0.3.2 Strand Notation

In early APL, a numeric vector constant was simply written as a list of numbers separated by spaces, for example `0 1 1 2 3 5` for the first six Fibonacci numbers. A major bone of contention in APL2 was the so-called *strand notation* (§4.2, §5.1), which extended this notation to non-constant vectors by juxtaposing array expressions. For example, the second expression below creates a 3-element vector of the scalar `'a'`, the 3-element vector `5 7 9`, and a 3-by-4 matrix:

```
      M←3 4⍴⍳12
      'a' (1 2 3+4 5 6) M
┌─┬─────┬──────────┐
│a│5 7 9│0 1  2  3 │
│ │     │4 5  6  7 │
│ │     │8 9 10 11 │
└─┴─────┴──────────┘
```

0.3.3 The Big Split

The community was split down the middle. One camp felt that strands were a natural and useful extension of constant vector notation; the other were convinced that it made the language harder to explain and implement, and that the construction of nested arrays should require explicit boxing and catenation. Unable to gain traction for his views at IBM, Iverson accepted an offer to join the SHARP APL language design group at I.P. Sharp Associates (IPSA) in 1980 [IPSA 1980].

The second-generation APL implementations produced a major fracture of the APL community into two incompatible dialects, one with floating arrays (APL2) and one with grounded arrays (SHARP APL). At STSC, the third major player at the time, Bob Smith implemented NARS, the Nested Array Research System, a system closely aligned with APL2 [Cheney 1981].

0.3.4 J and k

Only a few years later, the value of the mainframe-based APL market collapsed under the pressure of mini- and then microcomputers, which offered analytical applications much more memory and CPU. In 1987, IPSA was acquired by Reuters, who wanted the time series databases that IPSA had accumulated. Iverson retired from IPSA in 1987. In 1989, Iverson, together with Roger Hui and with input from Arthur Whitney, produced J, with a goal of providing a "shareware" APL implementation for use in teaching [Hui et al. 1990]. The special APL characters were abandoned because it was felt that they require technical solutions which at that time were still prohibitively expensive in an educational environment.

Inspired by work that he was doing on Wall Street, Arthur Whitney—who had worked with Iverson at IPSA [Iverson and Whitney 1982]—developed his own ideas about what an array language needed to do, creating A while at Morgan Stanley [Whitney 1989], and later the k language [Whitney 1993] and the Kdb+ array-based database engine in 2004 [Orth 2006; Whitney 2006a]. Designed for close integration with a high-volume, high-performance inverted column store, k added dictionaries and tables as native types, and a number of date and time types required for time series analysis.

J was clearly a "rationalization" of SHARP APL. k was a more radical departure but also has grounded arrays and no strand notation. Both J and k have acquired significant followings, and with the addition of the q query language [Whitney 2009] and Kx Systems partnering with First Derivatives plc, k has become a big commercial success. It is outside the scope of this paper to report further on these threads.

0.3.5 Dyalog APL

Although Dyalog APL will run most APL code written in 1983 when the language was first implemented based on the STSC NARS design [Cheney 1981], it can be argued that current Dyalog APL, with its support for object oriented programming, systematic treatment of arrays of objects (§4.3), "futures" for asynchronous programming (§4.4), tacit forms including J-style function trains (§1.3), and support for lexically scoped functional programming (§1.1), warrants classification of Dyalog APL as a descendant of APL, rather than a member of the APL2 family of dialects.

0.3.6 Convergence

A fiercely competitive era followed the split in which the relationships between players on all sides were strained. Gitte Christensen, chairman of the APL90 conference in Copenhagen, recalls that several members of the APL90 program committee objected to allowing Iverson to present J at that conference because it "was not APL" [Christensen 2006]. Conversely, in a public debate at the APL91 conference in Stanford, Donald McIntyre opined that an APL conference without Iverson was like "*Hamlet* without the Prince" [Barman and Camacho 1991]. The ISO Standard for Extended APL [ISO/IEC 2001], which at long last appeared due to the Herculean and Sisyphean efforts of Professor Lee Dickey, papered over a lot of cracks but failed to improve the situation. Finally, SHARP APL was withdrawn from the market and the split within the "core" APL community effectively disappeared.

In recent years, Bob Bernecky, Jim Brown, Eric Iverson, John Scholes, Bob Smith, and Arthur Whitney, as well as the authors, regularly exchange ideas on language design (joined by a number of younger people that you will hopefully hear from in a future HOPL conference). ACM SIGAPL, the ACM Special Interest Group on APL, re-interpreted the "APL" in its name in early 2009 as Array Programming Languages [WaybackMachine 2009], so that J, k, Nial, etc. would be included in its purview. (SIGAPL is now a chapter under ACM SIGPLAN.) Bob Smith's open source NARS2000 project [Smith 2020] and Dyalog APL have adopted some of the most significant ideas from SHARP APL and J into APL2-based core languages. A number of the language features that we discuss in more detail in the technical sections of this paper are in that category.

0.3.7 Open Source

As is the case for many popular programming languages, a number of APL dialects are available as free, "open source" implementations:

- J — Source available under GPL3 [Jsoftware 2018].
- NARS2000 — Source available under GPL3 [Smith 2020].
- NGN APL — Source code available at the GitHub NGN/APL project [Nickolov 2013].
- Nial — Designed and implemented by Michael Jenkins in 1983 in collaboration with Trenchard More [Jenkins 1989]. Source code available under GPL3 at the GitHub QNial7 project.
- GNU APL — Implements ISO APL with a large number of extensions [ISO/IEC 2001; Sauermann 2018].

For many modern programming languages, being open source has more or less become a requirement. This appears to be less critical for APL systems, possibly because the overlap between the potential users and implementers of APL systems is smaller than it is for most languages, where both groups are typically software engineers or computer scientists.

Most current APL users value the efforts of language vendors highly and have absolutely no desire to work on the implementation of the system. The same is fortunately not true for tools and libraries written *in* APL; these are starting to appear as open source projects as technology has made the sharing of APL source code easier.

It will be interesting to see whether future generations of APL users will be more technically savvy and simultaneously demand—and provide the resources required to implement—more competitive open source APL systems.

0.4 System Integration

APL is an executable mathematical notation, and Iverson saw his work as a continuation of an evolution that already spanned thousands of years [Iverson 1980]. The most important enhancements, that the bulk of this paper will be devoted to, are the improvements to the notation itself. However, typical APL implementers actually spend most of their time making sure that APL is well integrated with the popular software platforms of any given era—work on the notation is often a luxury to be indulged in, in peaceful moments. A number of these features have been instrumental in making APL a successful language for application development, and thus secured funding for continued work on the notation.

0.4.1 Shared Variables

In 1973, IBM released a major update to *APL*360, named APLSV [Falkoff and Iverson 1973a]. The "SV" stood for Shared Variables, which were shared between an APL session and an auxiliary processor which typically provided access to an external facility such the host file system. An APL program would share the variable, make a call by setting the shared variable, and then either monitor the state of the variable or, if synchronous mode had been selected, simply reference the variable (which would block until a response was available).

Shared variables became the standard mechanism not only for making foreign function calls from APL interpreters, but also to implement client/server systems by sharing variables between APL sessions. IPSA implemented support for network shared variables which were used for file transfer and inter-company e-mail over IPSANET [Potyok 1987; Moore 2005].

0.4.2 Component File Systems

One of the first additions to the APL*PLUS time-sharing system was the addition of a component file system (CFS). Any APL array can be written to a component file and subsequently retrieved using a component number. Virtually every APL interpreter subseqently implemented a variation of the original STSC/IPSA CFS. Some implementers saw the CFS as such an integral part of an APL interpreter that they made CFS look like primitives, using symbols ⍇ in place of the more traditional *system function* ("standard library") name `⎕fread` and ⍈ in place of `⎕freplace` (DEC APLSF, Burroughs APL/700, and MicroAPL APLX). IBM APL systems did not provide a full CFS, but STSC SHAREFILE was available as an add-on for many of the APL systems from IBM.

Most CFS systems support sharing of files, with fine-grained access controls, providing the backbone for highly secure multi-user systems like the IPSA and STSC e-mail systems [Goldsmith 2010], and time series, relational, and multi-dimensional data bases. Although the use of relational and other external databases has become more common with time, component files have provided a simple and efficient APL object store for many commercial applications for five decades.

0.4.3 Memory-Mapped Files

The use of "inverted" data structures, in which a table is viewed as a collection of columns rather than rows, has been a common practice since APL was invented. Each column becomes a simple (non-nested) array, which means that it is represented as a dense sequence of bytes addressable via computed offsets, rather than a collection of tuples each containing pointers to the individual column values. The result is a huge reduction in memory consumption and a corresponding speed-up for queries and updates. In an array language like APL, J, or k, there is no significant loss of convenience or expressive power when dealing with the lists in an inverted structure [Hui 2018b].

The memory mapping technology supported by modern operating systems, in which a real or virtual file appears to be memory, allows an array to be memory-mapped, bringing all the power of an array language to bear on potentially enormous data structures. Dyalog APL, J, and k all support the direct memory mapping of arrays. This technique is at the heart of the high performance of the Kdb+ database system (implemented in k in 2004) [Orth 2006; Whitney 2006b] and Jd (implemented in J) [Jsoftware 2017], and is used by many APL systems to achieve high performance analytics on large data sets.

0.4.4 Unicode

The Unicode standard [ISO/IEC 1993] has played an important role in making APL easy to use in conjunction with other tools. Thanks to the efforts of the APL Standards Committee, APL symbols were included in very early versions of the Unicode standard. Before the standard arrived, each individual APL system used its own set of 256 characters (or 512 characters in the case of DEC APL). As a result, not only was APL source code unreadable unless special APL-centric tools were used to display and edit it, all data moving in or out of most APL systems—or between different APL systems—needed to be translated.

It has taken a couple of decades for Unicode support to trickle out into all the nooks and crannies of operating systems and tool chains, and there are still occasional issues in browsers and on smartphones, but it is now the case that:

- You can read and write APL source code in a standard text editor on almost any platform.
- Linux systems ship with APL keyboard support out-of-the-box.
- Most source code management systems and "diffing tools" handle APL like any other language.
- It has become common for popular text editors to include simple syntax coloring for APL.

In other languages, it is common to use UTF-8 strings to represent Unicode text, but a variable length encoding makes array operations inefficient. Instead, many APL systems use multiple internal representations in order to efficiently support character arrays. Dyalog APL uses three representations for text, where each character consumes 1, 2, or 4 bytes depending on the required range of code points within the array [Dyalog 2018a, quad-ucs]. As with numbers, automatic type conversions happen under the covers. Most application data still only consumes a single byte per character, saving space and making searches run 2-4 times faster than they would if a single "wide" type had been used. However, the full range of Unicode can be used without type declarations or conversions.

0.4.5 The .NET Strategy

Microsoft's .NET Strategy, envisioned in the late 1990s, played a significant role in shaping the APL community at the start of the millennium. Microsoft's core vision was a "Common Language Runtime" (CLR), a portable virtual machine which would provide a set of "managed" data types that all language interpreters and compilers would be built upon [Gregory 2003]. Microsoft invited language implementers to participate in the specification work, including Dyalog Ltd. However, while there was sympathy from other language designers and occasional agreement over a beer that APL had the "best and most consistent" model for arrays, APL's view that an array was a value type rather than a reference type, that reference counting and copy-on-write were required to make large arrays perform well, and that scalars are simply zero-dimensional arrays rather than a set of distinct data types, was too much of a gap to bridge. The CLR did not adopt this definition of an array, and APL could not work well without it.

Effectively, APL interpreters had provided developers with a managed environment since the very first implementation in 1966—which contributed to the robustness and stability of early APL

time-sharing systems. There were fundamental semantic differences between the definition of CLR "managed" arrays and reference-counted APL arrays, whence an efficient APL interpreter would be forced to implement its own array data type rather than use core CLR types, and copy data on the fly. Since sharing data with other CLR languages was a main motivation for being on the CLR in the first place, this would make a new CLR-based APL unattractive. However, the managed-code propaganda machine was in overdrive and touted it as the only way to deal with buffer overruns and the threat posed by bandits and terrorists taking advantage of poor coding practices. For a while, the threat seemed very real that unmanaged applications would become very hard to install in client environments.

The pressure was particularly high in the USA, where some government contractors were told that they had limited time to produce managed solutions if they wanted to retain existing business. As a result, the APL+Win company, now named APL2000, decided that a managed APL interpreter was going to be required for the US domestic market, and the VisualAPL project was born. VisualAPL took advantage of Microsoft's Visual Studio Integration Program to build a system that allowed developers to switch in and out of APL from C#, based on infrastructure provided with Visual Studio.

Back in Europe, Dyalog decided to build a bridge which made it possible to consume and provide .NET classes, taking advantage of "reflection" to extract the metadata required to convert data on the fly between the APL environment and Microsoft's .NET Framework. The namespace and GUI class technology which Dyalog had developed for Microsoft's Component Object Model (COM) was extended to allow .NET objects to appear within the APL workspace. A notation for classes was developed, closely modelled after the capability of CLR classes. This allowed classes to derive from CLR base classes, support CLR interfaces, and to generate CLR definitions which in turn would allow any other CLR based language to consume or derive from Dyalog APL classes—even though the APL interpreter itself remained unmanaged.

One-way bridges, allowing APL users to take advantage of .NET components, were developed for MicroAPL's APLX, and eventually also for APL+Win, when it became clear that the majority of APL+Win users had decided not to upgrade to the new VisualAPL product.

Although the hype about managed code turned out to be exaggerated, the ability to integrate with software based on the .NET Framework did become critical with respect to continued use of APL in corporate environments. In particular, Dyalog's deep integration allowed the use of APL in the same way as any language on the .NET framework, allowing the use of APL as an ASP.NET scripting language and as an implementation language for SharePoint or Excel add-ins. This was instrumental in staving off forces that wished to cast APL as a technology from a bygone era.

At a time when Microsoft's Silverlight technology looked as if it would allow APL to run in browsers, Dyalog briefly flirted with the design and implementation of a prototype of a fully managed interpreter, with a project named APL#. Users were appalled at the idea of arrays that were reference types, and when Microsoft pulled the plug on Silverlight, APL# was quickly abandoned with a huge sigh of relief that the project never reached maturity and did not cause another split in the user base.

0.4.6 User Interfaces

APL is a notation that allows people to solve complex problems on a computer without having a traditional software engineering background. As interfaces become standardized and microservices become a popular delivery mechanism, it will be easier to use APL in combination with other technologies and free some APL users from having to implement their own user interfaces.

However, even in a world where interfaces are easy to use, the logistics of collaborating with others to create solutions using more than one technology presents significant challenges. Historically, the ability for a "non-technical" user to build a complete prototype of a solution in APL has been critical in bringing version 1.0 of many products to market. Providing UI building tools that are simple enough for an actuary to use, yet allowing the creation of sophisticated user interfaces, has been one of the most important success criteria for APL.

Some of the most successful tools for creating user interfaces in different eras have been:

- Mainframe: Auxiliary Processor AP124 provided state-of-the-art control over IBM terminals.
- MS-DOS: APL*PLUS/PC `⎕win` allowed complete control of the IBM PC's screen.
- MS-DOS and UNIX TTY: Dyalog APL `⎕sm` supported cross-platform UI on text-mode devices.
- MS Windows: Dyalog APL `⎕wc` allowed non-technical users to build Windows applications with a rich user experience.

More about this in §7.

0.5 APL Around the World

The .NET strategy section (§0.4.5) refers to differences between the approach taken by vendors in the USA and the UK. The relative market penetration of APL, and the uses that APL was put to, have also varied in different markets, although many of the earliest successful uses of APL were used by global corporations who used APL to collect and analyze data from across the globe.

At its inception, APL was most widely used in the USA and Canada. Over time, the highest market penetration probably occurred in Japan and Scandinavia [Saigusa 1994; Christensen 2014] where the relatively high cost of skilled labor made the efficiencies provided by APL more important, and the high value placed on consensus decisions made it more likely that supporters of radically different approaches to building IT systems would still find ways to collaborate to meet the needs of large organizations.

APL also has a following in the Soviet Union and Russia [Kondrashev and Luksha 1991]. The first APL machines in the Soviet Union arrived at the Computing Center of the USSR Academy of Sciences in 1975 in the form of two MCM/70 machines. Later, "APL was directly related to the design of the Soviet space shuttle *Buran*", used to model the manufacture of hull cover plates [Kondrashev and Luksha 1991, p.10]. The APL92 conference held in St. Petersburg brought Soviet APL users into the worldwide APL community [Shaw 1992]. Even today, the percentage of Russian k/q users is surprisingly high. Oleg Finkelshtyen and Pierre Kovalev, two young Russians who originally hailed from St. Petersburg, are implementers in the core team at Kx Systems.

0.6 APL Since 1978

The rest of this paper focusses on the history of APL since 1978 as a programming language. The history is organized along the same lines as would a presentation of the language, grouped by the main parts: user definitions, functions, operators (higher-order functions), arrays, syntax, and symbols, ordered so that notation is (mostly) defined before it is used. The story can be told in different ways; Appendix E provides pointers to several alternative histories.

0.6.1 APL in 1978

In 1978, arrays were homogeneous—all numbers or all characters—and simple—each array item was a number or a character (although they were not then called homogeneous or simple). Vector notation was available only for numeric or character constants (e.g., `3 1.4 1.5 ¯9265359` or `'Cogito, ergo sum.'`).

Primitive functions were classified as either scalar (as in scalar multiplication in conventional mathematical notation, but generalized to + - ÷ ⌈ ⌊ = ≠ etc.) or non-scalar AKA "mixed".

A user-defined function had fixed valence—monadic, dyadic, or niladic—and consists of numbered lines of code. The only form of flow control was → (goto). A user-defined function must be named (that is, could not be anonymous); the function name was embedded in line 0 along with specification of the arguments and results (if any) and the local variables. Local variables had dynamic scope. A function was defined either through mediation of the ∇ editor or by application of ⎕fx to a character matrix of the function lines.

There were seven operators (higher order functions): *reduce*, *reduce last*, *scan*, *scan last*, *inner product*, *outer product*, and *axis*. An operand for the first six must be a primitive scalar dyadic function; that for *axis* must be a function derived from one of the "slash operators" (*reduce*, *reduce last*, *scan*, or *scan last*) or one of a few mixed functions. The slash operators also accept a vector operand. Operators could not be defined by the user.

One realizes how arcane and arbitrary were the preceding only with the benefit of 40 years of hindsight.

> In any case, what is the point of saying about innovators that they should have done what later comers were able to do after the ground had been cleared for them?
>
> — Jacques Barzun, *From Dawn to Decadence* [Barzun 2000, p.46]

> Orville and Wilbur, why didn't you use a jet engine?
>
> — John Scholes [Scholes 2018, 9:02-9:08]

0.6.2 Personalities

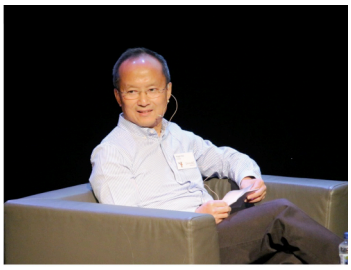
The design of APL since 1978 received significant input from a small number of people with intertwined personal relationships.

Ken Iverson led the way with his seminal paper *Operators and Functions* [Iverson 1978b]. The ideas were further developed in *Practical Uses of a Model of APL* [Iverson and Whitney 1982], *Rationalized APL* [Iverson 1983b], *A Dictionary of APL* [Iverson 1987], and *J Introduction and Dictionary* [Hui and Iverson 2004].

Iverson and Eoin Whitney (father of Arthur Whitney) were both born in 1920 in small towns in northern Alberta, Canada, both served in the RCAF in WWII, and both won graduate scholarships to Harvard in the early 1950s, where they first met and became friends [Whitney 2006b]. In 1969, Iverson showed APL to 11-year-old Arthur Whitney [Cantrill 2009], and in 1974 recommended him for a summer student position at IPSA Calgary. In the early 1980s, Iverson and Whitney were on staff at IPSA Toronto and had a couple of years of fruitful collaboration [Iverson and Whitney 1982]. They kept in touch in subsequent years. Whitney's enduring and significant influence on APL (let alone on A and k and Kdb+) can be gauged by the number of instances in this paper of "invented by Whitney", "on a suggestion by Whitney", and the like.

Roger Hui and Whitney were undergraduates at the University of Alberta in the 1970s and became acquainted in Professor Muldowney's real analysis class. Hui was a summer student at IPSA Calgary in 1975 where Whitney was the year before. In the mid 1980s, Iverson and Hui were employed at IPSA Toronto; the seeds of their collaboration were planted by the paper *Some Uses of { and }* [Hui 1987]. When Iverson and Whitney met to discuss APL at the Kiln Farm one summer weekend in 1989, Hui was present on the first day by Whitney's invitation. The "one page thing" (§10.2) that Whitney wrote on the afternoon of the second day provided the final impetus that got J started. Between 1989 and 2004, Hui apprenticed with Iverson working on J.

Ken Iverson & Arthur Whitney, 1989



Roger Hui, 2018

Morten Kromberg started at IPSA in the spring or summer of 1979 at age 16, when he was given a key to the IPSA Oslo office and a free APL account. In 2005, he became CTO and later CXO of Dyalog Limited (in a sense the guerrilla (*q.v.*) who took over command of the army), together with Gitte Christensen who assumed the role of CEO. Coming from a background in SHARP APL and taking charge of an APL2-based language implementation, Christensen and Kromberg have been instrumental in healing some of the wounds opened when second-generation APL systems were born. Kromberg independently invented futures and isolates (§4.4).

Morten Kromberg, 2016

Gitte Christensen, 2018

An important factor in these personal relationships was the worldwide e-mail system Mailbox (`666 box`) extant at IPSA since the early 1970s [Goldsmith 1980; Goldsmith 2010]. Mailbox provided for discussion groups on language design, programming questions, newbie help, etc. as well as person-to-person exchanges. (Kromberg (in Norway) first "met" Christensen (in Denmark) through Mailbox. They are still together.) IBM developed its own e-mail system after a key developer (Jon McGrew) used Mailbox and consulted with its implementer [McGrew 2016].

Jim Brown first encountered APL in 1965 when he started work at IBM Federal Systems in Owego, New York. In 1969, he got a summer position with the APL group at IBM Yorktown Research. Due to an administrative error, the summer job was not terminated after the summer and lasted three years [Brown 1971, 2016]. He shepherded APL2 through IBM from conception to product release. The key innovations are nested arrays and strand notation, respectively general arrays and generalized vector notation [Brown 1984]. Brown left IBM in 1996.

Jim Brown, 1982

Bob Smith encountered APL at the National Security Agency in 1969/70 [Hui 2005b; Smith 2018]. He joined STSC in 1971. In 1981, he designed and implemented NARS, officially to do nested arrays but included many of the ideas in *Operators and Functions* [Iverson 1978b; Cheney 1981]. He left STSC in 1983 but returned to APL in 2006 to develop NARS2000 [Smith 2018].

Bob Smith, 2016

John Scholes started with Atkins Computing and Sigma APL. At Dyalog, he and colleagues developed Dyalog APL, intending to follow APL2 but using NARS as the blueprint—information about APL2 did not become publicly available until after Dyalog APL version 1.0 in April 1983. Scholes later designed and implemented direct functions (§1.2) [Scholes 1996, 2018; Wikipedia 2019b].

John Scholes, 2018

John Daintree joined Dyalog Ltd. in September 1991, and designed and implemented objects and namespaces [Dyalog 2008a]. He became "the GUI guy" and later Chief Architect.

John Daintree, 2018

Bob Bernecky joined IPSA in 1971 [Bernecky 2016]. He was an implementer of SHARP APL and in the 1980s was also the manager of the IPSA language design and implementation group. He worked with Iverson and others in the transition to a second-generation APL [Bernecky and Iverson 1980, Bernecky et al. 1983, Bernecky 1987]. From 1990 his work has focussed on APL compilers [Bernecky et al. 1990; Bernecky 1997].

Bob Bernecky, 2015

0.6.3 Guerilla Projects

It is striking how many of the ideas important to current APL were nurtured in "guerrilla projects", projects undertaken with the grudging approval, or tolerance, or ignorance, or outright objection, of management. These took place coeval with rapid changes in the "APL biz" and in the ecology of computing.

APL2 developed over 14 years, an unusually long gestation period, with coding starting in 1971 and release as an IBM program product in 1984. In that time, the developers were kept busy "producing other results of value to the company", and were transferred between three or four divisions and moved between the US Atlantic and Pacific coasts [Falkoff 1991; Brown 2016].

Operators and Functions [Iverson 1978b] was published while Iverson was at IBM. Unable to gain traction for his views at IBM, Iverson accepted an offer to join IPSA in 1980 [IPSA 1980]. Further development of the ideas gained impetus in the early 1980s when Iverson and Arthur Whitney found themselves employed at IPSA Toronto at the same time. Officially, Whitney was supposed to be working on the OAG (Official Airline Guide) database, and he did, rewriting it from scratch and turning a failed project for a critical customer into a resounding success; unofficially, Whitney collaborated with Iverson and implemented the new APL ideas in a model written in APL, spending much more time on the APL model than on OAG [Iverson 2016; Whitney 2016; Hui 2005b]. The work was reported in *Practical Uses of a Model of APL* [Iverson and Whitney 1982]. (While en route to the presentation of this paper at APL82 in Heidelberg, Whitney invented the *rank* operator (§3.1) and the leading axis emphasis (§2).) Iverson also participated in the OAG project. Eric Iverson, son of Ken Iverson and the IPSA manager of both Iverson and Whitney, was unaware of this participation until 35 years later [Iverson 2016; Hui 2005b].

Iverson continued to work on APL after he "retired from paid employment" in 1987 [Iverson 1991a]. The J project was launched when Whitney wrote a 1-page interpreter fragment one Sunday afternoon in 1989 (§10.2), and began in earnest when Roger Hui also retired from paid employment (or so he thought) to join forces with Iverson on 1990-04-01. The collaboration continued until Iverson's passing in October 2004.

The original goal of NARS [Cheney 1981] was to implement nested arrays, but many of the ideas in *Operators and Functions* were also implemented. Management supported the extra stuff "mostly by ignoring" Bob Smith, the designer and implementer, and "letting [him] do whatever [he] wanted". Smith left STSC in 1983 [Smith 2018].

John Scholes invented *direct functions* or dfns (pronounced "dee funs"), a major distinguishing advance of current APL over previous versions (§1.2). The ideas originated when he read a special issue of *The Computer Journal* on functional programming [Wadler et al. 1989]. He proceeded to study other publications on functional programming and became strongly motivated ("sick with desire", like Yeats) to bring these ideas to APL [Scholes 2018]. He initially operated in stealth because he was concerned the changes might be judged too radical and an unnecessary complication of the language; other observers say that he operated in stealth because the rest of Dyalog were not so enamored and thought he was wasting his time and causing trouble for people. Dfns were first presented in the Dyalog Vendor Forum at the APL96 conference and released in Dyalog APL in early 1997 [Scholes 1996]. Acceptance and recognition were slow in coming. As late as 2008, in *Dyalog at 25* [Dyalog 2008a], a publication celebrating the 25th anniversary of Dyalog Ltd, dfns were barely mentioned (mentioned twice as "dynamic functions" and without any elaboration).

Even *APL\360*, the original APL, never received the full-throated backing of IBM. Fred Brooks was the discussant of the HOPL I paper *The Evolution of APL* [Falkoff and Iverson 1978]. In his remarks, he self-described as a midwife of APL and PL/I and spoke of the effects of the "one-language policy" at IBM in the 1960s:

> An outgrowth of the development of PL/I was an enunciation of an attempt to take on FORTRAN, ALGOL, and COBOL and replace them all with PL/I—what was known as the "one-language policy". These languages each had sufficiently strong community groups. The one-language policy had no noticeable effect on them. The only one that it had any effect on that I could see was unfortunately the implementation of APL that was underway at the time. So it's rather like one draws his sword and goes out to slay the foe, and in the process of drawing it back, clobbers the friend in the head!

0.6.4 The Five Ws

Operators and Functions [Iverson 1978b] substantially foretold what APL is today. From that and from the other general array ideas [Cheney 1981; Brown 1984] one can discern the *why* of many of the APL language features introduced since 1978: generalization, extending existing ideas and constructs; unification, making sense of mixed functions; composition, combining functions to derive new functions; rationalization/simplification; and exploitation, taking advantage of changes in hardware and software. These are illustrated below as a 5-dimensional sparse array—the *why* and the *what* are the horizontal and vertical axes; the other three axes, the *who*, *when*, and *where*, are flattened in the display.

Table 1: The Five Ws

		generalization	unification	composition	rationalization	exploitation
§4.2	general arrays	a,b,c,f				
§4.2	strands	a,c,f				
§4.1, §6.5	data types	b,n				
§3.1	rank	b,d,j	b		d,j	
§2	leading axis	e,i,j			f	
§3	operators			b,c,j		
§1.3	trains			g,j		
§1.1	control structures				h	
§1.2	direct functions				k	
§4.3	objects					l
§0.4.4, §6.3.4	Unicode					m
§4.4	futures & isolates					o

a	1971	Brown	IBM	[Brown 1971]
b	1978	Iverson	IBM & IPSA	[Iverson 1978b]
c	1981	Smith	STSC	[Cheney 1981]
d	1982	Whitney	-	[Whitney 2004]
e	1982	Whitney	-	[Bernecky 1987]
f	1984	Brown	IBM	[Brown 1984]
g	1988	Iverson & McDonnell	IPSA	[Iverson and McDonnell 1989]
h	1989	Whitney	Morgan Stanley	[Whitney 1989]
i	1992	Whitney	-	[Whitney 1992]
j	1992	Hui & Iverson	Jsoftware	[Hui and Iverson 2004]
k	1996	Scholes	Dyalog	[Scholes 1996]
l	1997	Daintree	Dyalog	[Kromberg 2007]
m	2008	Dyalog	Dyalog	[Dyalog 2008b]
n	2011	Smith	NARS2000	[Smith 2020]
o	2014	Kromberg	Dyalog	[Dyalog 2014a]

About the APL Examples

In the examples, a user entry is indented three spaces; its result is displayed except when the leftmost action is assignment. The symbol ⍝ indicates that the rest of the line is a comment. For example:

```
      ⍳ 5       ⍝ the function ⍳ (indices) applied to 5
0 1 2 3 4
      a←⍳5      ⍝ no display since leftmost action is assignment
      ⊢ b←⍳5    ⍝ ⊢ is the identity function
0 1 2 3 4
```

Sometimes, examples are formatted "2-up" or "3-up" across the page for a more compact display.

```
      * ⍳3  ⍝ exponential
1 2.71828 7.38906

      2 * 0 ¯1 2  ⍝ power
1 0.5 4

      6 + 2 * 0 ¯1 2
7 6.5 10

      (6 + 2) * 0 ¯1 2
1 0.125 64
```

```
      'W' = 'WWKD'  ⍝ Boolean vector result
1 1 0 0

      ⍝ sum derived from operator ≠ (reduce)
      +≠ 'W'='WWKD'
2
      sum ← +≠
      sum 1 6 15 20 15 6 1
64
      *≠ 2 2 2 1.37014335  ⍝ power tower
64
```

The examples above show that numeric vector constants (`0 ¯1 2`) are entered with the items juxtaposed, separated by blanks. Similarly, character vector constants (`'WWKD'`) have the items juxtaposed, enclosed in quotes.

The examples on the left illustrate that APL functions are ambivalent, with a monadic and a dyadic definition yoked to the same symbol, in this case `*`. Functions apply to array argument(s) and produce array results (scalars are 0-dimensional arrays). Functions are evaluated from right to left; the same rule applies when parentheses are used, but (as usual) a parenthesized expression must be completely evaluated before its result can be used. Floating-point results are displayed to 6 decimal digits of precision (but are computed with IEEE 754-2008 binary64 numbers).

The examples on the right illustrate deriving a function from an operator (a higher-order function), and then applying it to an argument.

APL definitions in the paper use the symbols ⍺, ⍵, and ↔: ⍺ denotes the left argument, ⍵ the right argument, and ↔ (from conventional mathematical notation) denotes equivalence. For example, the function `*` is defined as `*⍵ ↔ exp(⍵)` and `⍺*⍵ ↔ *⍵×⍟⍺` where `⍟⍵ ↔ ln(⍵)`.

Appendix B has a summary of the notation used in this paper; APL syntax is described in §5 and Appendix C; and expressions are executable in Dyalog APL [Dyalog 2018a] (unless noted otherwise) with the following environment settings:

```
⎕io ← 0      ⍝ index origin (§10.4)
⎕ml ← 1      ⍝ migration level
⎕pp ← 6      ⍝ print (display) precision
⎕div ← 1     ⍝ the result of 0÷0 is 0 [McDonnell 1976]
]box on      ⍝ display nested arrays using boxes (§4.2)
```

> The speed of modern computers has reached a point where, for many applications, the time required to program a problem may be a more important factor than the actual computing time. Attempts are made, therefore, to simplify the work of programming in various ways.
>
> — K.E. Iverson, Ph.D. Thesis, §2A4, [Iverson 1954b]

1 USER DEFINITIONS

1.1 Control Structures

In the beginning, up to 1978 and beyond, a user-defined function had numbered lines, starting from 0. The only form of flow control was → (goto), targeting a scalar number or the first of a vector of numbers, and → on an empty vector means "don't branch" (continue).

The limitations of goto can be mitigated by using line labels instead of absolute line numbers and by creative use of APL expressions. (The only exception to avoiding the use of absolute line numbers is →0, which means return from the current function.) The discipline is now so ingrained that typically functions are no longer displayed with line numbers.

For example, computing the hailstone sequence (Collatz conjecture) for positive integer n:

```
 z←h n
⍝ hailstone sequence
 →L10 ρ⍨ 1≠n ⋄ z←,1 ⋄ →0
L10:
 →L20 ρ⍨ 1≠2|n ⋄ z←n,h 1+3×n ⋄ →0
L20:
 z←n,h n÷2
```

```
 z←h1 n
⍝ hailstone sequence
 →((1=n),1 0=2|n)⌿one,odd,even
one:  z←,1 ⋄ →0
odd:  z←n,h1 1+3×n ⋄ →0
even: z←n,h1 n÷2
```

```
  h 9     ⍝ hailstone sequence for 9
9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

  h1 9    ⍝ ditto
9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

(The phrase ρ⍨ can be read as "if".)

Control structures for APL were "in the air" in the 1970s [Kelley 1972; Kelley 1973; Johnston 1977], but it was 1989 before they appeared in a major dialect: Whitney's A [Whitney 1989]. Since the early 1990s, control structures have been in all the major dialects but one (APL2). The hailstone sequence computation done with control structures:

```
 z←h2 n
⍝ hailstone sequence
 :If 1=n
   z←,1
 :ElseIf 2|n
   z←n,h2 1+3×n
 :Else
   z←n,h2 n÷2
 :EndIf
```

```
 z←h3 n
⍝ hailstone sequence
 :Select (1=n),2|n
   :Case 1 1 ⋄ z←,1
   :Case 0 1 ⋄ z←n,h3 1+3×n
   :Case 0 0 ⋄ z←n,h3 n÷2
 :EndSelect
```

1.2 Direct Functions

Iverson was dissatisfied with the way user functions were defined. (For example, for the hailstone sequence he would have had to write something like `h` or `h1`.) In 1974, he devised "formal function definition" or "direct definition" for use in exposition [Iverson 1974]. A direct definition has two or four parts, separated by colons:

```
name : expr
name : expr0 : prop : expr1
```

Within a direct definition, `⍺` denotes the left argument and `⍵` the right argument. In the first instance, the result of `expr` is the result of the function; in the second instance, the result of the function is that of `expr0` if `prop` evaluates to 0, or `expr1` if it evaluates to 1. Assignments within a direct definition are local. For example, the hailstone sequence as direct definitions:

```
h4: ,1 : 1<⍵ : h4a ⍵
  h4a: ⍵,h4 ⍵÷2 : 2|⍵ : ⍵,h4 1+3×⍵
```

Further examples of direct definition are available [Iverson 1980; Hui 1987].

Direct definition was too limited for use in larger systems. The ideas were further developed [Iverson 1978b, §8; Iverson and Wooster 1981; Cheney 1981, §4.17; Iverson 1983b; Iverson 1987; Bunda 1987; Hui et al. 1990] but the results were unwieldy. Of these, the "alternative APL function definition" of [Bunda 1987] came closest to current facilities, but is flawed in conflicts with existing symbols and in error handling which would have caused practical difficulties, and was never implemented. The main distillates from the different proposals were that (a) the function being defined is anonymous, with subsequent naming (if required) being effected by assignment; (b) the function is denoted by a symbol and thereby enables anonymous recursion.

A *direct function* [Scholes 1996] despite the name, is used to define functions *and* operators. It is commonly called a dfn, pronounced "dee fun". (Whence a function defined in the traditional manner, with control structures and all, is called a *tradfn*.) John Scholes designed dfns after studying a special issue of *The Computer Journal* on functional programming [Wadler et al. 1989] and other similar works, and implemented them over the objections of Dyalog colleagues. Currently, dfns are implemented in Dyalog APL, NARS2000 [Smith 2020] and others [Nickolov 2013]. They also play a key role in efforts to exploit the computational capabilities of a GPU [Hsu 2019].

A dfn is a sequence of possibly guarded expressions (or just a guard) between `{` and `}`, separated by `⋄` or new-lines.

```
expr
guard: expr
guard:
```

A guard must evaluate to a 0 or 1; its associated expression is evaluated if the guard is 1. A dfn terminates after the first unguarded expression which does not end in assignment, or after the first guarded expression whose guard evaluates to 1, or if there are no more expressions. The result of a dfn is that of the last evaluated expression. Dfns may be nested. Assignments in a dfn are local with lexical scope. (Dfns also have an error handling mechanism, not further described in this paper.)

Within a dfn, the following symbols have special meaning:

`⍺`	the left argument; `⍺←expr` is executed only if the dfn is invoked without a left argument
`⍵`	the right argument
`⍺⍺`	the left operand
`⍵⍵`	the right operand
`∇`	the function being defined (self-reference)
`∇∇`	the operator being defined (self-reference)
`⋄`	statement separator (same as new-line)

The hailstone sequence can be rendered as a dfn as follows.

```
h5←{
  1=⍵: ,1
  2|⍵: ⍵,∇ 1+3×⍵
       ⍵,∇ ⍵÷2
}

   h5 9     ⍝ hailstone sequence for 9
9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

   ≢ h5 9   ⍝ the number of items in this sequence
20

   ≢∘{1=⍵:,1 ⋄ 2|⍵:⍵,∇ 1+3×⍵ ⋄ ⍵,∇ ⍵÷2}¨ 1+3 10⍴⍳30
 1  2  8  3  6  9  17  4 20  7
15 10 10 18 18  5  13 21 21  8
 8 16 16 11 24 11 112 19 19 19
```

The last expression illustrates using an anonymous dfn as an operand and anonymous recursion: The right argument `1+3 10⍴⍳30` is a 3-by-10 matrix of the integers from 1 to 30. The *tally* function `≢` (§2.1) is composed with an anonymous dfn `{...}`, the same as `h5` except unnamed and expressed in one line with parts separated by the statement separator `⋄`, to compute the number of items in a hailstone sequence. That derived (composite) function is applied to each item of the right argument, mediated by the *each* operator `¨` (§3.6). The final result is therefore the hailstone sequence lengths for initial values from 1 to 30 as a 3-by-10 matrix. The same result obtains as `≢∘h5¨ 1+3 10⍴⍳30`.

Further examples of dfns can be found in §11 and in [Scholes 2019a; Hui 2016a; Wikipedia 2019a], and further explanations in [Wikipedia 2019a]. Dfns have been found to work well with operators. They are also used in "magic functions" §9.2 in the implementation of Dyalog APL.

It is interesting to note that most student participants in Dyalog's annual Problem Solving Contest, APL neophytes and Padawans all, choose to write in dfns rather in tradfns.

1.3 Trains

For years, Iverson struggled to achieve in APL the effect of `f+g` and `f×g` as they are written in calculus. From a "rigidity of viewpoint" and "wedded to the power of operators" [Iverson and McIntyre 2008], these attempts took the form of operators. (See §3.8 for the progression.) Finally, *trains* AKA *forks* were invented [Iverson and McDonnell 1989]. Previously, three functions in isolation were an error. The idea is to replace the error by the following interpretation:

```
⍺ (f g h) ⍵  ↔  (⍺ f ⍵) g (⍺ h ⍵)
  (f g h) ⍵  ↔  (f ⍵) g (h ⍵)
```

(↔ is a non-APL symbol denoting equivalence, as in conventional mathematical notation.)

Moreover, `(f ... h p q r)` ↔ `(f ... h (p q r))`, and an isolated sequence of *two* functions is also assigned a meaning (*atop*, described below), so that a train of any length, even or odd, is interpreted. Therefore, to write `f+g` and `f×g` as in calculus, you write `f+g` and `f×g` in APL. Iverson and Eugene McDonnell worked out the details during the long plane rides for the APL88 Conference in Sydney, Australia, with Iverson coming up with the initial idea on awaking from a nap [Hui 2004; Hodgkinson 2017].

Subsequently, it was realized that trains greatly increase the possibilities for "tacit definition", expressions consisting of compositions of functions which do not explicitly mention the arguments [Hui et al. 1991]. Trains are implemented in several dialects: J [Hui et al. 1991], NARS2000 [Smith 2020], NGN APL [Nickolov 2013], and Dyalog APL [Scholes 2013].

For example:

```
   ⊢ x ← 3 4 ⍴ 3 1 4 1 5 9 2 7 1 8 2 8
3 1 4 1
5 9 2 7
1 8 2 8
   (+⌿ ÷ ≢) x                +⌿x             ≢x          (+⌿x)÷≢x
3 6 2.66667 5.33333      9 18 8 16      3          3 6 2.66667 5.33333
   mean← +⌿ ÷ ≢
   mean x
3 6 2.66667 5.33333
   mean 3 1 4 1 5 9
3.83333
   mean ⍬                    +⌿⍬             ≢⍬          (+⌿⍬)÷≢⍬
0                         0               0          0
   3 (+,-,×,÷) 4
7 ¯1 12 0.75
```

The expressive completeness of trains depends on an *atop* composition of two functions:

```
    f Atop g ⍵  ↔  f g ⍵
⍺ f Atop g ⍵  ↔  f ⍺ g ⍵
```

Dyalog APL defines 2-trains as *atop*. That, together with the functions `⊣` (*left*) and `⊢` (*right*), allows many common compositions to be written as trains. For example:

`⍵ f ⍺`	↔	`⍺ (⊢ f ⊣) ⍵`	*commute* (passive voice)
`⍵ f ⍵`	↔	`  (⊢ f ⊣) ⍵`	*reflex*
`f ⍺ g ⍵`	↔	`⍺ (f g) ⍵`	*atop*
`f g ⍵`	↔	`  (f g) ⍵`	*atop*
`(g ⍺) f (g ⍵)`	↔	`⍺ ((g⊣) f (g⊢)) ⍵`	*over*
`(g ⍺) f (h ⍵)`	↔	`⍺ ((g⊣) f (h⊢)) ⍵`	
`⍺ f g ⍵`	↔	`⍺ (⊣ f (g⊢)) ⍵`	*hook* (*beside*)
`(g ⍺) f ⍵`	↔	`⍺ ((g⊣) f ⊢) ⍵`	*left hook*

Since the original formulation, trains have been extended: The isolated sequence of an array and two functions, is defined as follows [Hui 2005c]:

```
⍺ (a g h) ⍵  ↔  a g (⍺ h ⍵)
  (a g h) ⍵  ↔  a g (  h ⍵)
```

Or, equivalently, `a` is treated as a constant function in the general definition.

Trains are also used in the discussion on infinite arrays in §4.5.

Postscript. An operator solution for *fork* defied the best efforts of Iverson (and Whitney) for ten years. It turns out that an operator solution, although not general, *is* possible with an ingenious scheme where four "unlikely" left argument values are reserved for controlling function application [Last 2010].

1.4 Naming

Traditionally, the left-pointing arrow ← was used to assign names to arrays, and only arrays. A function was named by characters embedded in its representation (as in functions `h`, `h1`, `h2`, and `h3` in §1.1 and §1.2 above), and there was no direct way to name a derived function such as `+⌿`. In *Operators and Functions* [Iverson 1978b], a new symbol ⍤, ← overstruck with ¯ (§6.3.1), was proposed for naming functions. According to Iverson, the self-same ← would have been used but for Falkoff's strong objections [Iverson 2004].

The reservations were soon overcome and ← is now used to name arrays, functions, and operators alike [Bernecky and Iverson 1980; Iverson and Wooster 1981; Iverson and Whitney 1982].

2 FUNCTIONS

In modern APL, the design and use of APL functions are informed by an emphasis on operations on the leading axis of arrays. Traditionally, APL had separate versions for the leading axis and the trailing axis for several important functions and operators. The leading-axis functions can be used to model the trailing-axis ones with use of the *rank* operator ⍤ (§3.1), but not *vice versa*:

```
(⌽⍤1)⍵   ↔   ⌽⍵      reverse
⍺(⌽⍤1)⍵   ↔   ⍺⌽⍵     rotate (for scalar ⍺)
(f⌿⍤1)⍵   ↔   f/⍵     reduce (fold)
(f⍀⍤1)⍵   ↔   f\⍵     scan
```

Closely related to operations on the leading axis is the idea of major cells. A *major cell* of an array is a subarray with rank one less than the rank of an array arranged along its leading axis: an item of a vector, a row of a matrix, a plane of a 3-d array, etc. Operating on the leading axis is analogous to treating an array as a (conceptual) vector and with the major cells as its (conceptual) items. A function defined on major cells applies consistently to arrays of any rank by using the *rank* operator ⍤ (§3.1).

Once the importance of major cells and operating on the leading axis was realized, new functions were designed from the outset to work on major cells. Existing functions were extended to work on major cells, and the leading axis versions of existing functions, even when unextended, were emphasized over the trailing axis ones.

The leading-axis emphasis is consistent with prefix agreement (§3.1.3) for the *rank* operator. Internally, with row-major ordering for array storage, consecutive major cells occupy consecutive memory locations (and likewise the major cells of a major cell, all the way down), leading to more efficient execution (§9.1).

The leading-axis emphasis was invented by Whitney at the same time he invented the *rank* operator in July 1982. Iverson credited Whitney for using leading-axis functions to model replacements for the anomalous bracket axis operator [Iverson 1983b]; further attribution can be found in [Bernecky 1987]. The implications required some time to be understood: The term "major cell" appeared in the 1985-09-05 draft of [Iverson 1987] but not in [Iverson 1983b; Iverson et al. 1984] where it surely would have appeared if it had been in use then.

2.1 *Tally* ≢

In *A Programming Language* [Iverson 1962, §1.5], the function μ (Greek mu) is the number of rows of a matrix and the function ν (Greek nu) is the number of columns of a matrix or the number of elements of a vector. These were combined and generalized in *APL\360* into a single function ⍴, the *shape* of an array of any rank (dimensionality) as a vector result [Falkoff 1969]. (Falkoff quipped that he didn't know any other Greek letters.)

It is useful to have a function which returns the number of major cells in an array, as a scalar, the *tally* or *count* function ≢⍵. *Tally* was first implemented in A in 1989 [Whitney 1989], added to J on Whitney's suggestion [Hui et al. 1990], and more recently added to NARS2000 and Dyalog APL [Hui 2013c].

```
      ⊢ m←2 7⍴⍳14          ⍴m          ≢'deipnosophist'
0 1 2  3  4  5  6      2 7          13
7 8 9 10 11 12 13        ≢m          ≢3 1 4 1 5 9.265
                       2           6
```

The effects of having this simple function ≢ are subtle and far-reaching or, rather, the effects of *not* having this simple function are subtle and far-reaching. For example, for calculating the average of an array, {(+⌿⍵)÷≢⍵} is superior to {(+/⍵)÷⍴⍵} or {(+⌿⍵)÷⍴⍵}, and likewise +⌿÷≢ is superior to +/÷⍴ or +⌿÷⍴ [Hui 2010f].

One speculates that the permissive and infelicitous treatment of singles in the conformability rules for the primitive scalar dyadic functions, + − × ÷ ⌈, etc., arose out of a desire for arguments involving ⍴ of a vector to give a result instead of an error.

strict: (⍴⍺)≡⍴⍵, or ⍺ or ⍵ is scalar
permissive: (⍴⍺)≡⍴⍵, or ⍺ or ⍵ is single

(A *single* is an array all of which dimensions are 1s, ∧⌿1=⍴⍵.)

The permissive treatment insinuated itself into the inner product, and the *each* operator ¨ (§3.6) and the *rank* operator ⍤ (§3.1). Ideally you'd want f¨ and f⍤0 to be the same as f, where f is a primitive scalar dyadic function, and they are the same except when one argument is single. Then the decision is, do you want to be strict in ¨ and ⍤ and make them not the same, or do you want to be permissive so that they are the same? Different decisions have led to incompatibility between APL dialects:

	APL\360	Dyalog APL	NARS2000	APL2	J
scalar function	single	single	single	1-vector	prefix
each ¨	N/A	single	single	1-vector	prefix
rank ⍤	N/A	scalar	single	N/A	prefix

("1-vector" means one-element vector. "Prefix" is a generalization of "scalar", explained in §3.1.1, Agreement.)

Similarly, for A[i;j;k;...]←B, the conformability rules are:

strict: ⍴B ↔ (⍴i),(⍴j),(⍴k),... , or B is scalar
permissive: (⍴B)~1 ↔ ((⍴i),(⍴j),(⍴k),...)~1, or B is single

(⍺~1 removes 1s from vector ⍺.)

In fairness, the preceding smacks of sniping with 20-20 hindsight. Before *tally* came along, before 1978, an expression for the length of a vector as a scalar result would be something like:

```
''⍴⍴v
(⍴0)⍴⍴v
(⍳0)⍴⍴v
(⍴v)[0]
×/⍴v      ⍝ shortest and obscure
0⊥⍴v      ⍝ shortest and obscurest
```

With ≢, the expression is ≢v; absent ≢, the demand and temptation for permissive treatment of ⍴v, leading to permissive treatment of 1-element vectors, were probably overwhelming.

2.2 *Index-Of* ⍳ *et al.*

In *APL*\360, the *index-of* function ⍺⍳⍵ finds the index of the first occurrence in vector ⍺ of a scalar in array ⍵. In several dialects this has been extended to find major cells.

```
   3 1 4 1 6 ⍳ 3 5 1
0 5 1
   x
3 1 4 1 5 9
2 7 1 8 2 8
1 6 1 8 0 3
   'chthonic' ⍳ 'rhematic'
8 1 8 8 8 2 6 0
   a
chthonic
metonym
syzygy
```

```
   y
3 1 4 1 5 9
3 1 4 1 6 0
2 7 1 8 2 8
1 6 1 8 0 3
```

```
   b
syzygy
metonym
rhematic
dazlious
chthonic
```

```
   x ⍳ y
0 3 1 2
```

```
   a ⍳ b
2 1 3 3 0
```

```
   y ⍳ x
0 2 3
```

```
   b ⍳ a
4 1 0
```

Index-of is an instance of the searching problem, one well-known and well-studied in computer science. Some authors even wrote a book, well, half a book on the topic [Knuth 1973]. It has been the subject of discussion and study in APL as early as 1973 [Bernecky 1973] and as recently as 2017 [Hui 2010d, 2014, 2017b]. The extension of *index-of* to search for major cells is backward compatible.

The specific inner product `∧.=` has received optimization efforts since the 1970s [Bernecky 1977]. Nevertheless, the optimized implementation can be further sped up via the extended *index-of*: `⍺∧.=⍤⍵ ↔ (≠⍺)(↑∘.=↓)⍺⍳⍺⌿⍤⍵`, converting a cross-tabulation on vectors into an outer product on integers [Hui 2014, §12; Hui 2015a], and thence implemented as a magic function (§9.4).

Likewise, the *unique* function `∪⍵` was defined to find the unique items of a vector. It has been extended to find unique major cells.

```
   a
3 1 4 1 5 9
2 7 1 8 2 8
1 4 2 8 5 7
2 7 1 8 2 8
3 1 4 1 5 9
```

```
   ∪ 'eleemosynary'
elmosynar

   ∪ 'deontic'
deontic
```

```
   ∪ a
3 1 4 1 5 9
2 7 1 8 2 8
1 4 2 8 5 7
```

The closely-related *member* function `⍺∈⍵` computes whether a cell of array `⍺` is in array `⍵`.

```
   3 5 1 ∈ 3 1 4 1 6
1 0 1
```

```
   'chthonic' ∈ 'rhematic'
1 1 1 1 0 0 1 1
```

Originally, the cells to be sought were scalars. As for *index-of* and *unique*, an obvious extension would have been to look for major cells. Unfortunately, in current APL `⍺∈⍵ ↔ ⍺∈,⍵`, as though the right argument were ravelled (first made into a vector); therefore `∈` can not be extended and remain backward compatible. Therein is a lesson on the perils of permissive treatment of arguments, the lesson the more subtle and difficult if it is not realized that permissive treatment is taking place.

Suppose backward compatibility were not a requirement, and `∈` *could* be extended? There still are interesting things to say. For `∈`, the major cells are determined by the *right* argument; for `⍳` they are determined by the *left* argument. A hypothetical primitive `∋` would be more similar to `⍳`. Moreover, the order of the arguments in many primitive dyadic functions, when not constrained by firmly-established custom, is such that `a∘f` (currying a left argument) is a sensible monadic function.

2.3 *Interval Index* ⍸

In *interval index* `α⍸ω`, the major cells of `α` are required to be sorted and therefore partitions the domain into contiguous, half-open `[a,b)` intervals. `α⍸ω` finds the index of the interval which contains a cell of `ω`. For example:

```
      3 5 11 ⍸ 7 ¯5.1 3 11 19
1 ¯1 0 2 2

      'aeiou' ⍸ 'boustrophedonic'
0 3 4 3 3 3 3 3 1 1 0 3 2 2 0
```

Interval index was previously modelled [Hui 1987, §1.2; Hui 2016a, §14] and is similar to the *bins* function (denoted `α⌸ω`) in A+ [A+ 2003]. It was implemented in J [Hui 2005a] and more recently in Dyalog APL [Hui 2016d]. It provides an incentive for implementing a total array ordering [Brudzewsky et al. 2018; §2.5] whereby any array can be sorted.

2.4 *Index* `[]`

The *index* function embodies several issues in miniature, and examination of its evolution is illuminating. In *A Programming Language* [Iverson 1962, §1.5], element i of a vector was indicated by $\boldsymbol{v}_i$, row i of a matrix by $\boldsymbol{M}^i$, column j of a matrix by $\boldsymbol{M}_j$, and the entry at row i and column j of a matrix by $\boldsymbol{M}_j{}^i$. When *APL*\360 was implemented, the notation was linearized and rationalized:

- As was drolly noted [Falkoff 1969], the "corners" notation imposed an upper bound of 4 on the maximum rank array which can be indexed. The `A[i;j;k;...]` notation overcame this restriction.
- What is being indexed is an array (of course) but the indices themselves (the "subscripts") can also be arrays. For example [Hui 2016a, §4]:

```
      x← 3 1 4 1 5 9
      '.□'[x∘.>⍳⌈/x]
□□□.....
□.......
□□□□....
□.......
□□□□□...
□□□□□□□□□
```

```
      x ∘.> ⍳⌈/x
1 1 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0 0
1 1 1 1 1 1 1 1 1
```

  In the example, the 2-element character vector `'.□'` is indexed by a 6-by-9 Boolean matrix. Array indices form a Cartesian product of selections. Thus `A[2 1 3;4 0]` select the entries

```
A[2;4]   A[2;0]
A[1;4]   A[1;0]
A[3;4]   A[3;0]
```

  producing a result with shape `3 2`. In general,

  `ρA[i;j;k;...;y;z]` ↔ `(ρi),(ρj),(ρk),...,(ρy),(ρz)`.
- An elided index is interpreted to mean that every position along that axis is selected. Thus `A[;j]` selects columns `j` of `A`, `A[i;]` selects rows `i` of `A`, and `A[;]` selects `A` *in toto*.

When operators were contemplated [Iverson 1978b; Iverson 1983b; IBM 1994], shortcomings of the `[;]` notation became more evident: (a) it was difficult to write an indexing expression which works on an array of any rank; and (b) unlike other functions, there is no single function

symbol which can be used as an operand. The first problem is solved by the encoding made possible by general arrays: the phrase `i;j;k` where `i`, `j`, and `k` are integer arrays, can be written as a 3-element vector whose items are those arrays. In this encoding, many dialects abandoned elided indexing, and in the dialects which retained elided indexing the results are not overwhelmingly good. The second problem is solved by a new index function denoted by a single symbol, usually `⌷`, where `ix⌷A` indexes `A` with items of the vector `ix`, for example, `i j k⌷A ↔ A[i;j;k;;...;]`. This index function has many uses in diverse areas [Hui 1987, 2017a].

Arguably, *index* remains ... improvable. It is possible to provide abbreviated indexing, negative indexing, reach indexing, and scattered indexing in a single function `I`. The "kernel" is defined for a scalar left argument, and extends to higher-ranked left arguments per the *rank* operator `⍤` (§3.1); that is, `I ↔ Ik⍤0 ∞` for `Ik` defined on scalar left arguments.

- Abbreviated — Indices can be elided for trailing axes. Handy for indexing major cells.
- Negative — A negative index `i` is interpreted as `n+i` where `n` is the length of the axis. Thus `¯1` selects the last item.
- Reach — Index into nested arrays.
- Scattered — Comes "for free" with extension per the *rank* operator.

For example:

```
      x
┌─┬─┬─────────────────┐
│3│1│4                │
├─┼─┼─────────────────┤
│1│5│9                │
├─┼─┼─────────────────┤
│2│7│┌─────┬────┬───┐│
│ │ ││Cogito│ergo│sum│
│ │ │└─────┴────┴───┘│
├─┼─┼─────────────────┤
│8│2│8                │
└─┴─┴─────────────────┘
```

```
      0 3 1 0 I x
3 1 4
8 2 8
1 5 9
3 1 4
     ¯1 0 I x
8 2 8
3 1 4
     (⊂1 ¯1) I x
9
     (⊂2 2) I x
┌─────────────────┐
│┌─────┬────┬───┐│
│Cogito│ergo│sum│
│└─────┴────┴───┘│
└─────────────────┘
     (⊂(⊂2 2),2 ¯1) I x
m
```

```
      2 3 4⍴⍳24
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23

     (⊂1 2) I 2 3 4⍴⍳24
20 21 22 23
```

(The *enclose* function `⊂⍵` makes a scalar out of any array `⍵`.)

To date, no major dialect has such an *index*, although several come close, including the indexing used in the *at* operator `@` (§3.7).

Finally, a few words on typography. Some dialects use `{` to denote *index*. This is infelicitous because it conflicts with the use of the glyph for the even more valuable dfns (§1.2), and makes it appear that a matching `}` is missing. Other dialects use `⌷`, Unicode code point U+2337. It is nicely mnemonic (looks like I for *index*, get it?) and is likely a typographical, notational, and historical pun: `⌷` is `[` overstruck with `]` on the IBM 2741 terminal (§6.3.1).

2.5 *Grade* ⍋

Monadic ⍋⍵ produces the indices needed to sort ⍵ (thus (⊂⍋⍵)⌷⍵ sorts ⍵). It is stable—indices of equal items are in ascending order—and therefore can be used to model a lexicographic *grade* or *sort*. *Sort* obtains readily from *grade* and vice versa:

```
sort ← {⍵⌷⍨⊂⍋⍵}                    ⍝ sort from grade
grade← {1⌷⍤1 sort ⍵{⍺⍵}⍤¯1⍳≠⍵}     ⍝ grade from sort
```

For example:

```
   a                    sort a
to                   be
be                   be
or                   not
not                  or
to                   to
be                   to
```

```
   t←a {⍺⍵}⍤¯1 ⍳≠a   ⍝ ⍺⍵ is stranding (§0.3.2); ⍤ is rank (§3.1)
   t                    sort t                  1 ⌷⍤1 sort t
┌──┬─┐                ┌──┬─┐                  1 5 3 2 0 4
│to│0│                │be│1│
├──┼─┤                ├──┼─┤                     grade a
│be│1│                │be│5│                  1 5 3 2 0 4
├──┼─┤                ├──┼─┤
│or│2│                │not│3│
├──┼─┤                ├──┼─┤
│not│3│               │or│2│
├──┼─┤                ├──┼─┤
│to│4│                │to│0│
├──┼─┤                ├──┼─┤
│be│5│                │to│4│
└──┴─┘                └──┴─┘
```

Monadic ⍋ was originally defined only on numeric vectors, and was extended [Wooster 1980] to work on numeric arrays with higher rank. With that extension it has the distinction of being the first APL primitive function defined to work on major cells, before the term was invented or the importance of the concept realized. It was later extended to work on character arrays in Dyalog APL in 1982. More recently, ⍋ was extended in J to work with a TAO (total array ordering) [Hui 1996] on a suggestion by Whitney. TAO was taken up by Dyalog APL in 2018 [Brudzewsky et al. 2018]. The TAO also extends the left domain of ⍳. (The expression above for getting *grade* from *sort* requires TAO.)

Dyadic ⍺⍋⍵ was defined in 1979 and implemented in 1980 [Smith Jr. 1979; Wooster 1980] to order a character array ⍵ according to an alphabet (collating sequence) ⍺. In hindsight, it could have been simpler in definition and in use [Hui 2018a] and, more importantly, it preempted the alternative definition ⍺⍋⍵ ↔ ⍺⌷⍨⊂⍋⍵, index ⍺ by the grade of ⍵. Under the alternative definition, ⍋⍨⍵ ↔ ⍵⍋⍵ ↔ ⍵⌷⍨⊂⍋⍵, that is, sort ⍵, and with a TAO would sort any non-scalar array. Moreover, it can be argued that ordering by a collating sequence should never have been made a primitive function, due to the large variety and *ad hoc* variations in sequencing (accented characters, telephone book ordering, different natural languages, etc.).

2.6 Should It Be a Primitive?

APL has a large number of primitive functions, each denoted by a symbol (§6). How does one decide whether a function should be primitive? There does not exist a decision procedure which answers this question, an indication that language design is more an art than a science.

For example, the dialect J has the scalar function `j.` defined by `{⍺←0 ⋄ ⍺+⍵×0j1}¨`; that is, a scalar function whose monadic definition is `⍵×0j1` and whose dyadic definition is `⍺+⍵×0j1`. (The APL numeric constant ajb is $a + b \times \sqrt{-1}$ or $a + ib$ in conventional mathematical notation.)

If complex numbers are in the language, would you specify this as a primitive? A possible answer [Hui 2016c, §8]:

> Complex numbers can be constructed as ordered pairs of real numbers, similar to how integers can be constructed as ordered pairs of natural numbers and rational numbers as ordered pairs of integers. For complex numbers, `j.` plays the same role as `-` for integers and `÷` for rational numbers.

We don't know that this is *the* answer; Iverson designed the primitive and it was implemented without further discussion. We should have asked him about it.

3 OPERATORS

In APL, a *function* applies to array arguments to produce array results; an *operator* is a higher-order function in the sense of Heaviside, applying to function or array operands to derive a function result. An operator can be monadic or dyadic but not ambivalent.

APL's use of the terms "function" and "operator" differs from the usage in some programming languages, for example the C programming language, where a "binary operator" is a function denoted by a symbol using infix notation (e.g., `a-b`, `a>>b`); a "unary operator" is a function denoted by a symbol which uses prefix or suffix notation (e.g., `-b`, `b++`); and a "function" is denoted by a name, invoked by the name with the arguments enclosed in parentheses (e.g. `abs(a)`, `pow(x,y)`).

Operators are a main feature distinguishing modern APLs from the original APL. Their importance was recognized no later than 1973 (see §5 of *The Design of APL* [Falkoff and Iverson 1973b]). In 1978, there were seven operators, *reduce*, *reduce last*, *scan*, *scan last*, *inner product*, *outer product*, and *axis* [IBM 1975; Berry 1979]. An operand for the first six must be primitive scalar dyadic functions; that for *axis* must be a function derived from one of the "slash operators" (*reduce*, *reduce last*, *scan*, or *scan last*) or one of a few mixed functions. The slash operators also accept a vector operand.

The impetus for operators, and a guide for their development, came directly from *Operator and Functions* [Iverson 1978b], which specified 14 operators plus 21 "scalar operators" corresponding to the primitive scalar dyadic functions. Currently, some dialects have as many as 38 operators [Hui and Iverson 2004]. There are no *a priori* restrictions on operands; in particular, an operand can be a dfn.

The expressive power of operators can be seen as follows. An operator encapsulates extensive functionality in a single symbol. Since an operand can be a function (indicated below by `f` or `g`) or an array (indicated below by `a` or `b`) and the derived function is ambivalent, the following cases are possible:

monadic operator	dyadic operator
`(f MOP) ⍵`	`(f DOP g) ⍵`
`⍺ (f MOP) ⍵`	`⍺ (f DOP g) ⍵`
`(a MOP) ⍵`	`(f DOP b) ⍵`
`⍺ (a MOP) ⍵`	`⍺ (f DOP b) ⍵`
	`(a DOP g) ⍵`
	`⍺ (a DOP g) ⍵`
	`(a DOP b) ⍵`
	`⍺ (a DOP b) ⍵`

Operators economize the number of symbols needed to encode functionality. Suppose there are `n` symbols. If all of them denote functions, there are `2×n` possible functions (monadic and dyadic cases); if half denote functions and half monadic operators, there are

```
(2×n÷2)+2×(n÷2)*2  ↔  n+(n*2)÷2
```

possible functions; if half denote functions and half dyadic operators, there are

```
(2×n÷2)+2×(n÷2)*3  ↔  n+(n*3)÷4
```

possible functions. These are in addition to the possibilities afforded by array operands.

We now examine eight operators and one *non* operator.

3.1 *Rank* ⍤

Of all the operators introduced since the original APL, pride of place must be given to the *rank* operator, invented by Whitney on a train ride to the APL82 conference in Heidelberg, Germany in July 1982 [Bernecky 1987; Iverson 1991a; Whitney 2004; Pesch 2004] and first implemented in May 1983 [Bernecky et al. 1983]. It is a generalization of scalar extension, inner (matrix) product, and outer product in *APL*\360, `maplist` in LISP [McCarthy et al. 1959], `map` in modern functional programming languages, and the `broadcast` facility in NumPy [SciPy.org 2017]. Here, we describe the *rank* operator in detail because it is a microcosm of APL history.

In the beginning and up to the 1980s, APL had scalar functions (`+ - ⌈ ⌊` etc.) and "mixed" functions. Scalar functions are nice: subject to the inner and outer product dyadic operators, to the reduction monadic operators, to scalar extension, and are extended to higher-ranked array in a systematic manner; mixed functions are … mixed, subject to none of these things. [Iverson 1978b] provided a classification of APL functions in terms of their argument and result ranks. He defined *uniform functions* as those whose result shape depends only on the argument shapes (and therefore whose result rank depends only on the argument rank), and described how uniform functions are extended to higher-ranked arrays. The *rank* operator makes these ideas more useful and more powerful. Instead of a static classification, it is a dynamic facility which can be invoked with different operands and arguments.

The *rank* operator, together with functions having a leading axis (major cell) orientation (§2), provides nearly all of the functionality of the anomalous axis operator `f[a]` without its drawbacks [Bernecky 1987]: (a) *Rank* follows the same syntax as other operators; *axis* does not. (b) *Rank* applies uniformly to all functions; *axis* applies to some functions and not to others, and where it is applicable it does so in an *ad hoc* manner specific to each function.

Whitney's masterstroke did not arise in a vacuum but from a milieu in ferment, ripe for innovation. The progression can be seen in multiple references [Iverson 1978b, §6; Bernecky and Iverson 1980; Bernecky et al. 1983; Brown 1984, §20; Iverson 1987; Bernecky 1987; Hui et al. 1990; Hui and Iverson 2004; Hui 1995; ISO/IEC 2001, §9.3.3-5].

In brief, the *rank* operator is a means to apply a function to cells (subarrays) of the arguments. 0-cells (rank-0 cells) are scalars, 1-cells are vectors, 2-cells are matrices, and so on. For example:

```
   ⊢ m←3 3⍴⍳9 ⋄ v←10 20 30          v +⍤1 ⊢m    ⍝ 1-cells + 1-cells
0 1 2                             10 21 32
3 4 5                             13 24 35
6 7 8                             16 27 38

   +⌿ m      ⍝ sum                  v +⍤0 1 ⊢m  ⍝ 0-cells + 1-cells
9 12 15                           10 11 12
                                  23 24 25
   +⌿⍤1 ⊢m  ⍝ +⌿ on 1-cells        36 37 38
3 12 21
```

The following description is adapted from [Hui 1995].

3.1.1 Ranks

`f⍤r` derives a function which applies function `f` to argument cells of rank `r`, an integer scalar or vector with up to 3 integers of the monadic, left, and right ranks. `r` is used as `⍺3⍴⍵r` so that a single number specifies all three ranks, and two numbers specify the left and right ranks, with the second also specifying the monadic rank.

`r` specifies the maximum rank of an argument cell taken from the trailing axes; an argument with rank `r` or less is treated as a single cell. `r` may also be negative, in which case it specifies the number of leading axes to exclude from the cell shape. That is, the effective rank is `e←(⍴⍴⍵)⌊0⌈r+(0>r)×⍴⍴⍵`.

3.1.2 Frame and Cell

An effective rank `e` induces from the argument a (conceptual) array with shape `(-e)↑⍴⍵` (called the *frame*) of cells, each having shape `(-e)↓⍴⍵`. For example, suppose a rank-4 argument has shape `2 3 5 7`. The following table presents the various values for a given rank.

rank	effective rank	frame	cell shape
0	0	2 3 5 7	⍬
1	1	2 3 5	,7
2	2	2 3	5 7
3	3	,2	3 5 7
4	4	⍬	2 3 5 7
∞	4	⍬	2 3 5 7
¯1	3	,2	3 5 7
¯2	2	2 3	5 7
¯3	1	2 3 5	,7
¯4	0	2 3 5 7	⍬
¯∞	0	2 3 5 7	⍬

Effectively, a positive or zero rank specifies the number of trailing axes to take for the cell shape, and a negative rank specifies the number of leading axes to take for the frame.

An `r`-cell is a cell induced by rank `r`. `¯1`-cells are also called *major cells*, cells with rank one less than the rank of a non-scalar array arranged along its leading axis. Major cells play a key role in the language.

3.1.3 Agreement

In the dyadic case, two frames `lf` and `rf` are involved, from the left and right arguments. Several different treatments are possible:

- scalar agreement: `(lf≡rf)∨(lf≡⍬)∨(rf≡⍬)`, the left and right frames match, or one is the empty vector (`⍬`). If the frames match, there are an equal number of left and right cells, and the operand function applies to corresponding cells. If they do not match, one frame must be `⍬`, that is, there is one cell on one side, whence that one cell is applied against every cell on the other side. Scalar agreement is implemented in Dyalog APL [Dyalog 2015].
- prefix agreement: `(p↑lf)≡(p↑rf) ⊣ p←(≢lf)⌊(≢rf)`, one frame must be a prefix of the other. Let `ff` be the longer frame (that is, `ff←lf,p↓rf`). In this case a cell of the argument with the shorter frame is applied against `×⌿p↓ff` cells of the other argument. Prefix agreement is implemented in J [Hui and Iverson 2004], and is consistent with the emphasis on the leading axis (§2).
- suffix agreement: one frame must be a suffix of the other. J had suffix agreement before it switched to prefix agreement in 1992 on a suggestion by Whitney [Whitney 1992].
- strict agreement: the frames must match. No dialect has ever implemented this.

Prefix and suffix agreement are backward compatible (§10.3) extensions of scalar agreement.

For example, suppose `f⍤2 1` is applied to arguments with shape `2 3 4 5 6` and `2 3 7`:

`f⍤2 1`	left	right
argument shape	`2 3 4 5 6`	`2 3 7`
cell shape	`5 6`	`,7`
frame	`2 3 4`	`2 3`

With scalar agreement, an error would be signalled because the frames `2 3 4` and `2 3` do not match and neither is `⍬`. Similarly, with suffix and strict agreement, an error would be signalled. With prefix argument, the arguments would be accepted, because `2 3` is a prefix of `2 3 4`. Four cells of the left argument would be applied against each cell of the right argument.

After agreement, it makes sense to speak of *the* frame, which in the dyadic case is the longer of the left and right frames and in the monadic case is just the one frame.

3.1.4 Assembly

Individual results from the argument cells are assembled to produce the required final overall result. The individual results are first brought to a common maximal rank by prefacing unit axes to cells with lower rank, then brought to a common maximal shape by using *take* (`↑`) to pad with fills. The same-shape individual results are then assembled into an array with a shape of `ff,cms` where `ff` is the frame and `cms` is the common maximal shape.

Commonly, for "nice" functions, the uniform functions of [Iverson 1978b, §6], the individual results have the same shape without recourse to prefacing with unit axes or padding. In addition, the *enclose* function `⊂` (or the *box* function `<`) of §4.2 makes a scalar out of any array, and an operand of `(⊂f)`, *enclose* composed with `f`, produces individual cell results which are all scalars.

The assembly procedure can also be used on any operator or function having result cells with disparate shapes, to produce the overall result.

Monadic examples:

```
      ⊢ y← 0 10 ∘.+ 2 4⍴⍳8    ⍝ a rank-3 array
 0  1  2  3
 4  5  6  7

10 11 12 13
14 15 16 17
      ,y          ⍝ ravel y
0 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17
      ,⍤2 ⊢y      ⍝ ravel rank-2 cells (matrices) of y
 0  1  2  3  4  5  6  7
10 11 12 13 14 15 16 17
      ,⍤¯1 ⊢y     ⍝ ravel ¯1-cells of y (same as above)
 0  1  2  3  4  5  6  7
10 11 12 13 14 15 16 17
      ,⍤1 ⊢y      ⍝ ravel 1-cells (vectors) of y
 0  1  2  3
 4  5  6  7

10 11 12 13
14 15 16 17
```

The last example `,⍤1 ⊢y` is the identity function because *ravel* on a vector is the identity function. The same sequence, using *sum* instead of *ravel*:

```
   +⌿y          ⍝ sum y
10 12 14 16
18 20 22 24

   +⌿⍤2 ⊢y      ⍝ sum rank-2 cells (matrices) of y
 4  6  8 10
24 26 28 30

   +⌿⍤¯1 ⊢y     ⍝ sum ¯1-cells of y (same as above)
 4  6  8 10
24 26 28 30

   +⌿⍤1 ⊢y      ⍝ sum 1-cells (vectors) of y
 6 22
46 62
```

Dyadic examples:

```
   111 222 ,⍤0 2 ⊢y       ⍝ scalars (0) v matrices (2)
111  0  1  2  3
111  4  5  6  7

222 10 11 12 13
222 14 15 16 17


   111 222 ,⍤1 2 ⊢y       ⍝ vectors (1) v matrices (2)
111  0  1  2  3
222  4  5  6  7

111 10 11 12 13
222 14 15 16 17

   (2 2⍴111 222 333 444) ,⍤1⍤1 2 ⊢y
111 222  0  1  2  3
111 222  4  5  6  7

333 444 10 11 12 13
333 444 14 15 16 17
```

The last example, like the penultimate one, demonstrates vectors *v* matrices (`f⍤1 2`), but within that vectors are catenated to vectors (the `f` is `,⍤1`).

We asserted above that the *rank* operator provides a generalization of scalar extension, inner product, and outer product. These generalizations obtain as follows:

extension, monadic	`f⍤mf`	
extension, dyadic	`f⍤(lf,rf)`	
inner product	`(f⌿ g⍤(lg,¯1))⍤(1+lg,∞)`	↔ `f.g`
outer product	`f⍤(lf,rf)⍤(lf,∞)`	↔ `∘.f`

(`mf` is the monadic rank of `f`; `lf` and `rf` are its left and right ranks; etc.)

Extension is explicit in the description of the *rank* operator (e.g., see "scattered indexing" in §2.4), as is outer product, especially in the "agreement" part of that description. A special case of extension is *contraction*, which allows a function defined on high-ranked arguments to be applied to low-ranked cells in an array. For example, the *APL\360* definition of *index-of* is `⍳⍤1 0` (§2.2).

What of inner product? To make the definition easier to digest, first consider the specific case of inner product on `+` and `×`, with `⎕IO=0`, on two matrix arguments. The definition becomes `(+⌿ ×⍤0 ¯1)⍤1 ∞`. The ranks `1 ∞` specify that vectors (1-cells) of the left argument are applied against the right argument *in toto*, and in such application `×⍤0 ¯1` specifies the product of the scalars of those vectors against the rows (`¯1`-cells) of the right argument. Going from the specific `+` and `×` case to the general case: For the left argument, instead of a vector of items, think a (conceptual) vector of `⎕IO`-cells, subarrays with rank `⎕IO`. For the right argument, instead of rows, think major cells.

The following example demonstrates the intermediate steps in an inner product.

```
   ⊢ x←1+2 3⍴⍳6
1 2 3
4 5 6

   ⊢ y←10+3 4⍴⍳12
10 11 12 13
14 15 16 17
18 19 20 21

   ⍝ row 0 of x
   1 2 3 ×⍤0 ¯1 ⊢y
10 11 12 13
28 30 32 34
54 57 60 63
   +⌿ 1 2 3 ×⍤0 ¯1 ⊢y
92 98 104 110
   1 2 3 +⌿ ×⍤0 ¯1 ⊢y
92 98 104 110

   ⍝ row 1 of x
   4 5 6 ×⍤0 ¯1 ⊢y
 40  44  48  52
 70  75  80  85
108 114 120 126
   +⌿ 4 5 6 ×⍤0 ¯1 ⊢y
218 233 248 263
   4 5 6 +⌿ ×⍤0 ¯1 ⊢y
218 233 248 263

   x (+⌿ ×⍤0 ¯1)⍤1 99 ⊢y
 92  98 104 110
218 233 248 263

   x +.× y    ⍝ the primitive
 92  98 104 110
218 233 248 263
```

Conventionally, inner product proceeds in "row-by-column" order; the present inner product defined in terms of rank proceeds in "row-at-a-time" order, otherwise known as the CDC STAR inner product algorithm. "Row-at-a-time" is more efficient than "row-by-column" because (a) it has better cache characteristics; (b) it is able to exploit sparse left arguments; and (c) it is able to exploit left arguments from a small domain. More details can be found in [Hui and Iverson 2004; Iverson 1990; Bernecky 1997, p.87-91; Hui 2009; Hui 2020a, #IC2013]. One can compare this formulation with John Backus's matrix multiply program in FP [Backus 1978, §11.3.3]

```
Def MM = (αα IP)∘(αdistl)∘distr∘[1, trans∘2]
```

which is also functional, but is "row-by-column".

Abrams described a "general dyadic form"—a unified treatment of scalar, inner, and outer products [Abrams 1970, §II.E]. But the treatment is restricted to scalar functions, and performs extra computation from which the result obtains by application of dyadic transpose (§9.1).

Finally, the concept of *function rank* is closely related to the *rank* operator whereby each function is specified to have default ranks, as in the original classification [Iverson 1978b, §6]. A function then automatically extends to higher-ranked arguments without explicit invocation of the *rank* operator. Several dialects have a *rank* operator but with the automatic extension limited to the primitive scalar functions [Dyalog 2015].

3.2 *Power* ⍣

Power and *power limit* were defined in 1978 [Iverson 1978b, §2]. *Power* is a dyadic operator `f⍣n⊢ω` defined as follows:

`ω`	if `n=0`
`f⍣(n-1) ⊢f ω`	if `n` is a positive integer
`fi⍣(|n) ⊢ω`	if `n` is a negative integer, where `fi` ↔ `f⍣¯1` is the inverse of `f`

For example (using Newton iteration to estimate square root):

```
      {0.5×ω+2÷ω}⍣3 ⊢1
1.41422
      ⊢ r←{{0.5×ω+2÷ω}⍣ω ⊢1}¨ ⍳6
1 1.5 1.41667 1.41422 1.41421 1.41421
      2 - r×r
1 ¯0.25 ¯0.00694444 ¯0.0000060073 ¯4.51061E¯12 4.44089E¯16
```

The name and the symbol for the *power* operator are cognate with the power *function*, denoted by `*`. The *power* operator plays a role in computability theory similar to that played by the power set in set theory [§11.6; Hui 1992b; Hui 2016a, §39].

3.2.1 *Power Limit*

The right operand may also be a Boolean function: in `f⍣g ω` the left operand `f` applies repeatedly to `ω`, getting `y`, until `(f y)g y` is `1`. The result of `f⍣g ω` is then `f y`. For example:

```
      ⊢ r←{0.5×ω+2÷ω}⍣≡ 1
1.41421
      2 - r×r
4.44089E¯16
```

`SG ω` below is an example of using a left operand function with a non-scalar result. It computes the subgroup generated by the permutation(s) `ω` [Hui 1979; Hui 1987, §4.4].

```
      SG←{{∪(2 1*⍨⍴ω)⍴ω[;ω]}⍣≡(⍉⍪⍳⊃⊖⍴ω)⍪ω}

      1⊖⍳5
1 2 3 4 0
      SG 1⊖⍳5
0 1 2 3 4
1 2 3 4 0
2 3 4 0 1
3 4 0 1 2
4 0 1 2 3

      5 ⍴ 1 0,2⊖⍳5
1 0 2 3 4
      SG 5 ⍴ 1 0,2⊖⍳5
0 1 2 3 4
1 0 2 3 4

      g←{(2,ω)⍴(1⊖⍳ω),(1<ω)⌿1 0,2↓⍳ω}
      g¨ ⍳8
┌┬─┬───┬─────┬───────┬─────────┬───────────┬─────────────┐
││0│1 0│1 2 0│1 2 3 0│1 2 3 4 0│1 2 3 4 5 0│1 2 3 4 5 6 0│
││0│1 0│1 0 2│1 0 2 3│1 0 2 3 4│1 0 2 3 4 5│1 0 2 3 4 5 6│
└┴─┴───┴─────┴───────┴─────────┴───────────┴─────────────┘
      ≢∘SG∘g¨ ⍳8
1 1 2 6 24 120 720 5040
      ! ⍳8
1 1 2 6 24 120 720 5040
```

In `SG`, prefacing `ω` by the identity permutation `⍳⊃⊖⍴ω` converts the limit computation into a closure computation. The examples illustrate that two permutations `(g ω)` suffice to generate the symmetric group [Herstein 1975, Ex. 2.10.11].

3.3 *Key* ⌸

Key is a monadic operator. In the dyadic case of the derived function `⍺ f⌸ ⍵`, major cells of `⍺` specify keys for the corresponding major cells of `⍵`, and `f` is applied to each unique key in `⍺` and the selection of cells in `⍵` having that key. In the monadic case of the derived function, `f⌸⍵` ↔ `⍵ f⌸ ⍳≠⍵: f` is applied to each unique key in `⍵` and the indices in `⍵` of that key.

Key was defined and implemented in J in 1989 or 1990 [Hui 2007] and in Dyalog APL in 2015 [Dyalog 2015; Hui 2020b]. It is motivated by the "generalized beta" operation in The Connection Machine [Hillis 1985, §2.6], but generalizes the generalized beta by accepting arrays of any rank, not just vectors, and by permitting any function, not just reductions (folds). *Key* is also cognate with the GROUP BY statement in SQL. For example:

```
   x←'Mississippi'

   {⊂⍺}⌸x      {⊂⍵}⌸x                    {⍺⍵}⌸x
Misp        ┌─┬─────────┬─────┬─┐      ┌─┬─────────┐
            │0│1 4 7 10 │2 3 5 6│8 9│  │M│0        │
            └─┴─────────┴─────┴─┘      ├─┼─────────┤
                                       │i│1 4 7 10 │
                                       ├─┼─────────┤
                                       │s│2 3 5 6  │
                                       ├─┼─────────┤
                                       │p│8 9      │
                                       └─┴─────────┘
```

The following snippet solves a "Programming Pearls" puzzle [Bentley 1983]: given a dictionary of English words, here represented as the character matrix `a`, find all sets of anagrams.

```
   a           {⍵[⍋⍵]}⍤1 ⊢a          ({⍵[⍋⍵]}⍤1 {⊂⍵}⌸ ⊢) a
pats        apst                  ┌────┬────┬────┐
spat        apst                  │pats│teas│star│
teas        aest                  │spat│sate│    │
sate        aest                  │taps│etas│    │
taps        apst                  │past│seat│    │
etas        aest                  │    │eats│    │
past        apst                  │    │tase│    │
seat        aest                  │    │east│    │
eats        aest                  │    │seta│    │
tase        aest                  └────┴────┴────┘
star        arst
east        aest
seta        aest
```

The algorithm works by sorting the rows individually (note the use of the *rank* operator `⍤` (§3.1)), and these sorted rows are used as keys ("signatures" in the Programming Pearls description) to group the rows of the matrix. As the anagram example illustrates, other APL functions can be used to create the requisite keys. The following is an example of *interval index* `⍸` (§2.3) and `⌸` working together to illustrate the central limit theorem, that the sum of independent random variables converges to the normal distribution [Hui and Iverson 2004; Hui 2016b, §F].

```
      t←¯1+{≠⍵}⌸(⍳51),(4×⍳50)⍸+⌿?10 1e6⍴21
      5 10⍴t
       0      0      0      0      2      3     16     54    112    301
     676   1330   2483   4459   7181  11315  16997  24017  32858  42405
   52898  62668  71031  77489  81804  81873  78669  72805  64705  55205
   45223  35131  26170  18899  12546   8083   4834   2934   1521    720
     353    153     47     23      5      1      0      1      0      0
```

`t` counts the number of occurrences for interval endpoints `4×⍳50`, of `1e6` samples from the sum of ten repetitions of uniform random selection of the integers from 0 to 20. A barchart of `t`:

```
      '.□'[(6e3×⍳⌈((⌈/t)÷6e3)∘.≤t]
.........................□□□.........................
.......................□□□□□.......................
.....................□□□□□□□.....................
...................□□□□□□□□□...................
.....................□□□□□□□.....................
...................□□□□□□□□□...................
.................□□□□□□□□□□□.................
...................□□□□□□□□□...................
.................□□□□□□□□□□□□□.................
...............□□□□□□□□□□□□□□□...............
.................□□□□□□□□□□□□□.................
...............□□□□□□□□□□□□□□□...............
.............□□□□□□□□□□□□□□□□□□□.............
...........□□□□□□□□□□□□□□□□□□□□□□□...........
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

The derived function `{≠⍵}⌸x` counts the number of occurrences of each unique cell of `x`. The Dyalog APL and J implementations recognize particular useful operands for *key*, for example `{≠⍵}` and `{f≠⍵}`, and implement those cases with special code (§9.3) for better performance. When `⌸` was introduced to Dyalog APL in 2013 it was discovered that `{≠⍵}⌸x` (*tally* of unique elements) ran almost as fast as finding the maximum of `x` [Hui 2013b]. Investigation into how that came about led to the following programming puzzle [Hui 2014, §16]:

> Find the maximum of a vector of 1-byte ints without using multicore, vector instructions, loop unrolling, etc. Can you do it faster in C than the following code snippet?
>
> ```
> max=*x++; for(i=1;i<n;++i){if(max<*x)max=*x; ++x;}
> ```

The puzzle stumped some expert C programmers. It is possible to be faster by a factor of 1.5.

3.4 *Stencil* ⌺

Stencil is also known as *cut*, *tessellate*, or *tile*; it was introduced by Iverson [Iverson 1983b, §K] and implemented in J [Hui and Iverson 2004] and Dyalog APL [Hui 2016d; Hui 2017d; Hui 2020c]. *Stencil* is a dyadic operator `f⌺s⊢⍵` which applies `f` to (usually overlapping) rectangles in `⍵`. The sizes of the rectangle and its movement are controlled by the vector or matrix `s`, where:

- Rectangle sizes are specified independently for each axis. A (trailing) elided size is assumed to be the length of the axis.
- Each element (or pair of elements for even size) of `⍵` in its turn is the middle of a rectangle.
- `f` is applied to a right argument with uniform shape, padded with fills as necessary to have the shape specified by `s`, and a vector left argument of the number of fills in each axis.
- Movements are 1 by default, and can be specified explicitly in `s[1;]`.

```
   ⊢ x←4 5⍴1 2 3 4 5 6 7 8 9
1 2 3 4 5
6 7 8 9 1
2 3 4 5 6
7 8 9 1 2
   {⊂⍵}⌺3 5⊢x    ⍝ enclose each rectangle
┌─────────┬─────────┬─────────┬─────────┬─────────┐
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
│0 0 1 2 3│0 1 2 3 4│1 2 3 4 5│2 3 4 5 0│3 4 5 0 0│
│0 0 6 7 8│0 6 7 8 9│6 7 8 9 1│7 8 9 1 0│8 9 1 0 0│
├─────────┼─────────┼─────────┼─────────┼─────────┤
│0 0 1 2 3│0 1 2 3 4│1 2 3 4 5│2 3 4 5 0│3 4 5 0 0│
│0 0 6 7 8│0 6 7 8 9│6 7 8 9 1│7 8 9 1 0│8 9 1 0 0│
│0 0 2 3 4│0 2 3 4 5│2 3 4 5 6│3 4 5 6 0│4 5 6 0 0│
├─────────┼─────────┼─────────┼─────────┼─────────┤
│0 0 6 7 8│0 6 7 8 9│6 7 8 9 1│7 8 9 1 0│8 9 1 0 0│
│0 0 2 3 4│0 2 3 4 5│2 3 4 5 6│3 4 5 6 0│4 5 6 0 0│
│0 0 7 8 9│0 7 8 9 1│7 8 9 1 2│8 9 1 2 0│9 1 2 0 0│
├─────────┼─────────┼─────────┼─────────┼─────────┤
│0 0 2 3 4│0 2 3 4 5│2 3 4 5 6│3 4 5 6 0│4 5 6 0 0│
│0 0 7 8 9│0 7 8 9 1│7 8 9 1 2│8 9 1 2 0│9 1 2 0 0│
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
└─────────┴─────────┴─────────┴─────────┴─────────┘
   {+⌿,⍵}⌺3 5 ⊢x    ⍝ sum the elements of each rectangle
27 40 46 39 30
36 54 66 57 45
54 69 78 63 45
33 39 47 38 27
   ⊢ A←3 5⍴ 0 1 2 1 0  1 2 4.5 2 1    ⍝ weights
0 1 2   1 0
1 2 4.5 2 1
0 1 2   1 0
   {+⌿,A×⍵}⌺3 5 ⊢x  ⍝ weighted sum for each rectangle
30.5 49    63.5  63   44.5
60   88.5 103   101.5 61.5
60   90.5 101    85.5 57
63.5 81    83.5  54.5 37
   {⊂⍵}⌺5⊢'abcdef'
┌─────┬─────┬─────┬─────┬─────┬─────┐
│  abc│ abcd│abcde│bcdef│cdef │def  │
└─────┴─────┴─────┴─────┴─────┴─────┘
   {⊂⍺⍵}⌺5⊢'abcdef'
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│┌─┬─────┐│┌─┬─────┐│┌─┬─────┐│┌─┬─────┐│┌──┬─────┐│┌──┬─────┐│
│├─┼─────┤│├─┼─────┤│├─┼─────┤│├─┼─────┤│├──┼─────┤│├──┼─────┤│
│└─┴─────┘│└─┴─────┘│└─┴─────┘│└─┴─────┘│└──┴─────┘│└──┴─────┘│
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘
```

An example of John Conway's Game of Life [Gardner 1970] is obligatory with this operator. `life` below is due to Jay Foad, translated from an algorithm in k by Whitney [Hui 2017c]. It applies the rules of the Game of Life to the universe to create the next generation.

```
      life ← {3=s-ω∧4=s←{+⌿,ω}⌺3 3⊢ω}

      ⊢ glider←5 5⍴0 0 1 0 0 1 0 1 0 0 0 1 1,12⍴0
0 0 1 0 0
1 0 1 0 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

      life glider
0 1 0 0 0
0 0 1 1 0
0 1 1 0 0
0 0 0 0 0
0 0 0 0 0

      {'.⍟'[⍵]}⍣¨ (⍳8) {life⍣⍺⊢⍵}⍣¨ ⊂glider
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│..⍟..│.⍟...│..⍟..│.....│.....│.....│.....│.....│
│⍟.⍟..│..⍟⍟.│...⍟.│.⍟.⍟.│...⍟.│..⍟..│...⍟.│.....│
│.⍟⍟..│.⍟⍟..│.⍟⍟⍟.│..⍟⍟.│.⍟.⍟.│...⍟⍟│....⍟│..⍟.⍟│
│.....│.....│.....│..⍟..│..⍟⍟.│..⍟⍟.│..⍟⍟⍟│...⍟⍟│
│.....│.....│.....│.....│.....│.....│.....│...⍟.│
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
```

In `life`, the derived function `{+⌿,ω}⌺3 3` computes the sum of each 3-by-3 square, moving by 1 in each dimension. The function `{'.⍟'[⍵]}` produces a compact display for a Boolean array. `⍣` is the power operator (§3.2).

3.5 *Under* `⍢` and *Obverse* `⍤`

Under was defined in 1978 [Iverson 1978b, §8], partly implemented in SHARP APL in 1981 [Iverson 1981], and fully implemented in J in 1990 [Hui et al. 1990]. *Obverse* was implemented in J in 1990 [Hui and Iverson 2004]. *Under* and *obverse* have been proposed for Dyalog APL [Foad 2017].

The dyadic operator *under* `f⍢g` is defined as

```
   f⍢g ⍵  ↔  gi f g ⍵
⍺ f⍢g ⍵  ↔  gi (g ⍺) f (g ⍵)
```

where `gi` is the inverse of `g` (that is, `gi` is `g⍣¯1`.) *Under* elucidates the important but often mysterious concept of duality in mathematics: Duality is *with respect to* some function `g`, that is, *under* `g`. For example:

```
      3 +⍢⍟ 4 2 7.5                  3 -⍢⍟ 4 2 7.5
12 6 22.5                      0.75 1.5 0.4

      3 × 4 2 7.5                    3 ÷ 4 2 7.5
12 6 22.5                      0.75 1.5 0.4
```

The expression `+⍢⍟` and `-⍢⍟`, plus under log and minus under log, capture the essential idea which underlay the design of slide rules.

```
   mean ← +⌿ ÷ ≢

   mean 3 1 4 1 5 9      ⍝ arithmetic mean
3.83333
   mean⍤⊛ 3 1 4 1 5 9    ⍝ geometric mean
2.85364
```

Examples of *under* are found not only in mathematics but abound in everyday life [Hui 2005e]. The "under anaesthetics" example provides a graphic illustration. Several steps are composed:

apply anaesthetics
  make surgical incision
    do procedure
  apply surgical suture
wake up from anaesthetics

The inverse steps are pretty important! The "pipe laying" example provides another illustration: dig a trench, lay the pipe, cover the trench. Finally, a more poetic example: ashes to ashes, dust to dust.

A striking duality is the `×⍤FFT` algorithm for fast digital multiplication. Both `FFT` (fast Fourier transform) and its inverse `IFT` are $O(n \times \circledast n)$, and `×` (scalar multiplication) is $O(n)$, so the whole expression is $O(n \times \circledast n)$, better than the $O(n \star 2)$ "school" method.

While `FFT` and `IFT` have terse and beautiful definitions in APL [Hui 2016a, §32], it'd be difficult for the system to recognize that they are an inverse pair. The *obverse* operator `⍤` solves the problem: `f⍤g ↔ f` except that `f⍤g⍣¯1 ↔ g⍤f`. Therefore, fast digital multiplication is, in full, `×⍤(FFT⍤IFT)`.

3.6 *Each* ¨

Each (¨) is a monadic operator such that `f¨ ↔ f⍤⊃⍤0`, a form of which was presented in 1978 [Iverson 1978b, §10]. It is the one new primitive operator in APL2 [Brown 1984]. *Each* is similar to *map* in modern functional programming languages but applies to arrays of any rank, not just lists. The *rank* operator (§3.1) is a generalization of *each*.

Primitive scalar functions such as `+ × ⌈ =` are "nice" because, in the dyadic case:

- Scalar extension: a scalar argument is applied against each scalar in the other argument and the shape of the result is the shape of that other argument.
- Otherwise, the argument shapes must be the same, and the function applies to corresponding scalars. The result shape is the same as the argument shape.

In the monadic case, the function applies to each item and the result shape is the same as the argument shape.

`f¨` confers all of the above properties on any function `f`.

For historical reasons, there is an anomaly where an argument which is *single* (all elements of the shape are 1) is treated as if it were a scalar, and further special treatment if *both* arguments are single. We avoid talking about that when we can get away with it. See §2.1 and §10.3 for further thoughts on the topic.

3.7 *At* @

`@` is a dyadic operator. All four combinations of array and functions operands are defined; they all merge the left operand with the right argument `⍵` at the items selected by the right operand [Dyalog 2018a; Scholes 2019b]. The array-array combination `a@b⊢⍵`, expected to be the most common, can be read as "`a` at `b` of `⍵`".

We posit an index function `J←{1≥≡⍺:⍺[]⍤0 ∞⊢⍵ ⋄ ⍵[⍺]}` on indices `⍺` and array `⍵`, which selects major cells of `⍵` if `⍺` is simple (not nested), or select an item or select into a nested array otherwise (§2.4). The operator `@` is then defined as follows:

```
  a@b ⊢⍵  ↔  W ⊣ (b J W)←   a          ⊣ W←⍵
  f@b ⊢⍵  ↔  W ⊣ (b J W)←   f (b J W) ⊣ W←⍵
⍺ f@b ⊢⍵  ↔  W ⊣ (b J W)← ⍺ f (b J W) ⊣ W←⍵
```

The temporary variable `W` illustrates that the right argument `⍵` is not modified. Instead, it is merged with `a` or `f a` or `⍺ f a`, at `b`, to create a new array result.

The right operand may be a boolean function which is applied to the right argument `⍵` to produce a boolean mask. The ravelled mask selects items from the ravelled `⍵` to be merged.

```
  a@g ⊢⍵  ↔  W ⊣ (m≠,W)←   a          ⊣ m←,g W←⍵
  f@g ⊢⍵  ↔  W ⊣ (m≠,W)←   f (m≠,W) ⊣ m←,g W←⍵
⍺ f@g ⊢⍵  ↔  W ⊣ (m≠,W)← ⍺ f (m≠,W) ⊣ m←,g W←⍵
```

For example:

```
      x
┌─┬─┬─────────────┐
│3│1│4            │
├─┼─┼─────────────┤
│1│5│9            │
├─┼─┼─────────────┤
│2│7│┌─────┬───┬──┐│
│ │ ││Cogito│ergo│sum│
│ │ │└─────┴───┴──┘│
├─┼─┼─────────────┤
│8│2│8            │
└─┴─┴─────────────┘
```

```
      2 7 1@2 ⊢x
3 1 4
1 5 9
2 7 1
8 2 8
```

```
      (2 3⍴10+⍳6)@0 2⊢x
13 14 15
 1  5  9
10 11 12
 8  2  8
```

```
      11@(⊂2 2) ⊢x
3 1  4
1 5  9
2 7 11
8 2  8
```

```
      '!'@(0≠≡¨)x
3 1 4
1 5 9
2 7 !
8 2 8
```

```
      ⊖@0 2⊢x
┌─┬─┬─────────────┐
│2│7│┌─────┬───┬──┐│
│ │ ││Cogito│ergo│sum│
│ │ │└─────┴───┴──┘│
├─┼─┼─────────────┤
│1│5│9            │
├─┼─┼─────────────┤
│3│1│4            │
├─┼─┼─────────────┤
│8│2│8            │
└─┴─┴─────────────┘
```

```
      ,⍤⍤⍤0'!'@(⊂2 2)⊢x
┌─┬─┬─────────────┐
│3│1│4            │
├─┼─┼─────────────┤
│1│5│9            │
├─┼─┼─────────────┤
│2│7│┌─────┬───┬──┐│
│ │ ││Cogito!│ergo!│sum!│
│ │ │└─────┴───┴──┘│
├─┼─┼─────────────┤
│8│2│8            │
└─┴─┴─────────────┘
```

At is the latest in a series of attempts at a "functional" equivalent of indexed assignment. The first attempt [Pesch 1981] was a *monadic* operator `⌶` with an array operand deriving a function, whose monadic case `a⌶⍵` is *index* and whose dyadic case `⍺(a⌶)⍵` is a merge of `⍺` at items `a` of `⍵`. The

monadic case was not taken up because *index* (§2.4) is better as a function: In `a⌶ω` the operand `a` is not subject to other operators such as *rank*, and the operator `⌶` can not be used as an operand to other operators. The dyadic case with variations was soon taken up [Iverson and Whitney 1982; Iverson 1983b; Iverson 1987; Hui and Iverson 2004]. Notably, it was proposed [Iverson and Whitney 1982] that if `α S ω` is a dyadic selection function on `ω` with "indices" (or some other parameter) `α`, then the dyadic case of `α∘S` can be defined to do *merge*, for example, `a(α∘⍞)ω ↔ a@α⊢ω`.

The various *at* operators are cognate with the *mesh* and *mask* operators in *A Programming Language* [Iverson 1962, p.19-21]. For boolean vector `u` and vectors `a` and `b`,

mesh: if `c←\`***`a`***`,`***`u`***`,`***`b`***`\`, then `a≡(~u)≠c` and `b=u≠c` ↔ `b@(u∘⍳) (~u)↓a`
mask: if `c←/`***`a`***`,`***`u`***`,`***`b`***`/`, then `((~u)≠c)≡(~u)≠a` and `(u≠c)≡u≠b` ↔ `(u≠b)@(u∘⍳) a`

The expressions `\`***`a`***`,`***`u`***`,`***`b`***`\` and `/`***`a`***`,`***`u`***`,`***`b`***`/` are in Iverson notation from 1962, not executable in APL.

The fitness of an *at* operator depends crucially on the fitness of the *index* function. Since *index* is improvable (§2.4), so too *at* is improvable.

3.8 Trains Encore

In this, the last subsection on operators, we revisit the evolution of a language facility which turned out *not* to be an operator. As described in §1.3, for years Iverson struggled to achieve in APL the effect of `f+g` and `f×g` as written in calculus. From a "rigidity of viewpoint" and "wedded to the power of operators" [Iverson and McIntyre 2008], these attempts took the form of operators:

- *Scalar operators* [Iverson 1978b, §4]: for each primitive scalar dyadic function, a scalar operator is defined denoted by the function symbol overstruck (§6.3.1) with the macron (overbar): `f⨪g` and `f⨯g`.
- *Til* operator [Iverson and Whitney 1982; Iverson 1983a,b], defined as follows:
  ```
    f T g ω  ↔  (g ω) f ω
  α f T g ω  ↔  (g ω) f α
  ```
  (*Til* was denoted `⍨` in [Iverson and Whitney 1982] and `}` in [Iverson 1983a,b]. These were never implemented and conflict with current usage. `T` is used here to avoid confusion.)
  The significance of *til* is that,

monadic	dyadic	
`f T g T h ω`	`α f T g T h ω`	
`(f T g) T h ω`	`α (f T g) T h ω`	operator syntax
`(h ω) (f T g) ω`	`(h ω) (f T g) α`	definition of *til*
`(g ω) f (h ω)`	`(g α) f (h ω)`	definition of *til*

  As in *Notation as a Tool of Thought* [Iverson 1980], in the two derivations above each line is equivalent to the line below it, for the reason given on the far right.
- *Catenation* operator [Iverson et al. 1984]:
  ```
  α f COP g ω  ↔  (α f ω),[⍳¯1](α g ω)
    f COP g ω  ↔  (f ω),[⍳¯1](g ω)
  ```
- *Union* and *intersection* operators [Iverson 1987]:
  ```
  f U g ω  ↔  (f ω),[⍳¯1](g ω)
  f I g ω  ↔  g ⌿ f ω
  ```
- *Yoke* operator [Iverson 1988]: Here, `:` is a monadic operator which results in a dyadic operator, and:
  ```
  α f g: h ω  ↔  (f α) g (h ω)
    f g: h ω  ↔  (f ω) g (h ω)
  ```

Finally, *trains*. See §1.3 for the rest of that story.

4 ARRAYS

Given the central role of arrays in APL, one might ask, what is an array? One answer [Hui 2012]:

> An *array* is a function from a set of indices to items of numbers, characters, … . A rank-*n* array is one whose function *f* applies to *n*-tuples of non-negative integers. A rank-*n* array is *rectangular* if there exist non-negative integer maxima $\boldsymbol{s} = (\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_{n-1})$ such that $f(\boldsymbol{i}_0, \boldsymbol{i}_1, \ldots, \boldsymbol{i}_{n-1})$ is defined (has a value) for all integer $\boldsymbol{i}_j$ where $(0 \le \boldsymbol{i}_j) \wedge (\boldsymbol{i}_j < \boldsymbol{s}_j)$. $\boldsymbol{s}$ is called the *shape* of the array.

Most APL implementations use reference counting and copy-on-write to avoid copying arrays passed as arguments to functions or operators or which become embedded in nested structures.

4.1 Simple Homogeneous Arrays

A simple homogeneous array is one whose items are all scalar characters or all scalar numbers.

In the beginning, numbers could be bits, integers, and floating-point numbers. Complex numbers were added in the 1980s [Penfield 1979; Penfield 1981; McDonnell 1981a]. The transition among different kinds of numbers is automatic so that in practice it is useful to think of abstract numbers, and of numeric arrays rather than integer or floating-point arrays. In some dialects, numbers can also be extended precision integers and rationals (J, NARS2000), extended precision floats (NARS2000), and quaternions and octonions (NARS2000) [Hui and Iverson 2004; Smith 2020].

Boolean arrays merit comment. In APL, the result of a proposition is 1 or 0 rather than true or false, so that it will be in the domain of arithmetic functions. This allows terse computations such as `+⌿x>100` for the number of items of vector `x` which are greater than 100. Knuth calls this "Iverson's convention" or "Iverson bracket" and has used it in his writing [Knuth 1992; Graham et al. 1989]. Algebraic manipulations of mathematical formulas using Iverson brackets look familiar to APL programmers because they resemble refinements of APL expressions. APL bit arrays also enables use of masking as a form of flow control.

Boolean arrays require 1 bit per item and are therefore very efficient in space. Recent enhancements to computer architectures (e.g., the BMI extensions to the x86 instruction set architecture) greatly increased the efficiency of bit array manipulations [Intel 2019; Lochbaum 2017]. Likewise, the 1- or 2-byte integer data types in Dyalog APL offer efficiencies in space and time. It should be possible, for example, to sort 2-byte integers faster than the same number of 4-byte integers. See also §9.2 Small-Range Data.

Like numbers, characters come in various sizes (1-, 2-, or 4-bytes), including ASCII and Unicode characters, and also like numbers, the transition among them is automatic so that it is sensible to think of them as "abstract characters".

4.2 Nested and Boxed Arrays

A nested (boxed) array is one where at least one item is other than a single number or a single character. (It is possible for a 0-item array to be nested [Brown 1984].)

With the sound and fury over nested *v* boxed, floating *v* grounded arrays (§0.3), it is instructive to examine the similarities and differences between them:

Table 2: Nested *v* Boxed Arrays

nested (floating) arrays [Brown 1971, 1984]	boxed (grounded) arrays [Iverson 1978b, §10; Iverson 1987]
function `⊂` *enclose*	function `<` *box*
the result of `⊂⍵` is a scalar	the result of `<⍵` is a scalar
`⍵≡⊂⍵` if `⍵` is a scalar number or character `⍵≢⊂⍵` otherwise	`⍵≢<⍵` for all `⍵`
no new data type	new data type
primitive scalar functions are pervasive	no primitive functions are pervasive
inverse function `⊃` *disclose* (denoted `↑` in Dyalog APL and called *mix*)	inverse function `>` *open*
a nested array can be produced by: • explicit application of *enclose* • explicit application of the inverse of *disclose*, `⊃⍤¯1` (§3.2) or `f⍨⊃` (§3.5) • implicit application of *enclose* through functions and operators defined in terms of same—*reduction*, *inner product*, *outer product*, *each*, etc. • strand notation	a boxed array can be produced by: • explicit application of *box* • explicit application of the inverse of *open*, `>⍤¯1` or `f⍨>`

To avoid confusion the functions `<` and `>` are called *box* and *open* here, even though prior to 1987 they were also called *enclose* and *disclose* [Iverson 1987].

It is possible to have *enclose* and have *reduction*, *inner product*, and *outer product* not defined in terms of *enclose*. For example, the ISO APL standard speaks of **Enclose-Reduction-Style** and **Insert-Reduction-Style** [ISO/IEC 2001, §9.2.1]. But in floating array systems, including Dyalog APL, these functions *are* defined in terms of *enclose*.

The two systems are incompatible. However, they offer similar functionality and expressiveness and application code can be written in either system without much difficulty. This was also the conclusion drawn in a preliminary analysis [Orth 1981].

4.3 Objects

As Iverson and colleagues were creating APL in the 1960s, Ole-Johan Dahl and Kristen Nygaard laid the foundations of what was to become the object oriented (OO) paradigm. Interestingly, although Simula was radically different from APL, Dahl and Nygaard were also searching for ways to describe and model complex systems [Nygaard and Dahl 1978].

Until the arrival of Microsoft Windows 3.1 and widespread graphical user interfaces in 1992, APL users remained blissfully unaware of object orientation. By the mid-90s, OO had become a popular way to describe graphical user interfaces, and by the end of the decade that followed, OO had become the dominant paradigm for all types of application programming interfaces. Major platforms like Sun's Java [Gosling et al. 2015] and Microsoft's .NET [Microsoft 2017] evolved, based on the idea that virtually everything was an "object".

In order to provide users with access to APIs and frameworks, APL language designers searched for ways to integrate into APL, where everything is an array, selected aspects of the OO paradigm, where everything is an object. Much of the most successful work was done by new arrivals in the market, and a wide variety of solutions resulted with no attempt at standardization.

Some interpreters, like APL+Win, avoided incorporating objects into the APL heap (or "workspace"). In APL+Win, the interface to external objects like GUI forms or TCP sockets is based

on passing strings containing class and object names to library functions, without the objects appearing as values in the workspace.

Other systems added features which allowed variables, functions and operators to be organized in dynamic objects in the workspace. The resulting containers are known as *namespaces* (Dyalog APL) and *locales* (J). k implements *dictionaries*, which are also containers for arrays, but are typically used to contain arrays representing the columns of a relational table. APLX, Dyalog APL, and VisualAPL added language extensions which allowed the definition and instantiation of actual classes. The same language features which supported namespaces or classes within APL were used to wrap the external objects used by object oriented APIs, or platforms like the Microsoft's OLE and .NET frameworks.

The choice of syntax for referring to a member `name` of an object `emp` was not straightforward. The traditional `emp.name` used in most OO languages was problematic in that the syntax was already in use for the generalized inner product, denoted `α f.g ω`. The ambiguity is resolved by interpreting `emp.name` as an object name reference and not an inner product, if `emp` is a namespace.

Most APL systems did adopt the notation from other OO languages—with a few exceptions.

`emp.name`	in most APL systems
`name__emp`	in J (retaining right-to-left evaluation)
`emp[`name]`	in k (using a *symbol* within index brackets to select from a dictionary)

The notation `emp.name` still leaves an important question open: is `emp` restricted to representing a single object, or can it be an array containing multiple objects? The .NET based VisualAPL is the only dialect that defines arrays as actual objects, exposing array metadata such as the type, rank, and shape as properties. At the other end of the spectrum, Dyalog APL and APLX see arrays as a higher level of organization: an array can *contain objects* but the array itself *is not an object*. Thus, `emp.name` is an array of references, having the same shape as `emp`, containing the value of the `name` member from each object contained within `emp`, which can be an array of any rank and shape. The items of `emp` do not have to have the same type, but they must all expose a `name` property or an error will be signalled.

This idea is not unique to APL; in SQL [ISO/IEC 2011], `emp.name` also refers to the `name` property of every record in the `emp` table. k dictionaries are also similar to relational tables, and the language provides extensive support for relational table queries.

In Dyalog APL, the dot can be followed by an expression in parentheses, in which case the expression is executed in the context of each element of the array to the left of the dot. For example:

```
emp.(name,': ',dept)
```

will return an array of the same shape as `emp`, with an enclosed character vector containing the name and dept for each employee, separated by the constant `': '`. A single name to the right of the dot is treated as a special case which does not require parentheses.

When evaluating a compound expression with multiple dots, evaluation proceeds from left to right. At each dot, the expression to the right is executed in the context of the array thus far produced. When single objects are involved, the effect is unsurprising to a user of any object oriented language which uses dots. The following expressions create an instance `XL` of the OleClient class `Excel.Application`, and `XL.Workbooks.Open` is an invocation of the `Open` method of the `Workbooks` class, returning a reference to the opened workbook, which is assigned to `wb`:

```
XL←⎕NEW 'OleClient' (⊂'ClassName' 'Excel.Application')
wb←XL.Workbooks.Open⊂'//some/workbook.xlsx'
```

If the evaluation of the segments left of a dot has produced an array of object references, the name or expression to the right of the dot is applied to each item of this array. If this array is nested, evaluation is done on the leaves of the array ("pervasively"). Continuing with `wb` from the previous example, one can extract the first two rows of every sheet in an Excel workbook as a nested vector with an element corresponding to each worksheet, each containing a two-row matrix of values, as follows:

```
wb.(⍞Sheets).UsedRange.(2↑Value2)
```

`wb.Sheets` is a collection. The monadic use of the *index* function `⍞` is called "materialize", which returns arrays unchanged, but converts enumerable collections into arrays through iteration. Thus, `wb.(⍞Sheets).UsedRange` is list of references to all the `UsedRange` objects corresponding to the array of sheets. Finally, the expression `(2↑Value2)` extracts the first two rows of each `Value2` matrix. [Kromberg 2007] discusses the design of the Dyalog implementation of arrays containing objects.

Although APL interpreters have had extensive support for object oriented programming for nearly two decades, most APL users still feel that object and array paradigms are an awkward fit. Interpreted APL often struggles to perform well on large collections of small objects. Many of the benefits of OO are related to taking advantage of types, while much of the strength of the APL family is that you can write code which is *shape*, *rank*, and *type* agnostic—achieving many of the same goals as OO through radically different mechanisms.

A new generation of users coming to APL with object and typed functional backgrounds may change attitudes to the blending of objects, functions, and arrays in the decades to come. Compilers may reduce or remove the performance issues related to types (see §8). At the moment, objects are generally only used to access or provide external interfaces and only in rare occasions for algorithmic work. Nonetheless, the ability to connect smoothly with object oriented APIs and platforms has been a critical ingredient for the most successful APL systems.

4.4 Futures and Isolates

In a nested array system, each item of an array can be an array. Recent versions of Dyalog APL also allow an item to be an as yet uncomputed value known as a *future* [Dyalog 2016]. Many APL functions, such as *take* (`↑`), *transpose* (`⍉`), and *reshape* (`⍴`), create new arrays by selecting items from one or more existing arrays at the top level. These structural and selection functions can transform arrays containing futures without blocking. However, any primitive function which needs to use the actual value of the item (such as mathematical function like `÷`), will block until the value is known. Futures are reminiscent of "beating" and "drag-along" [Abrams 1970], which are concerned with executing APL expressions on a serial machine and simplifying APL expressions.

Futures were added to Dyalog APL together with a new type of namespace known as an *isolate*, a new function *isolate* (denoted `⍪`) which creates isolates, and a new operator *parallel* (denoted `⫴`). The function and the operator remain models written in APL, but futures and isolates are implemented as extensions to the interpreter.

An isolate appears to be a normal namespace within the active workspace, with all of its names visible and accessible using the normal `object.name` or `object.(expression)` syntax described in §4.3. However, the object actually resides in a separate process, completely isolated from the current application (hence the name). When an expression is invoked from outside an isolate, a future is returned immediately as execution begins inside the isolate. Futures can be assembled into arrays, reorganized using structural primitives, and passed as arguments to user-defined functions; when the actual value of a future is required by a primitive function, execution of that primitive function is suspended until computation of the value has completed.

The monadic *parallel* operator ⍎ derives a function which, when invoked, creates an empty isolate. If the operand function is user-defined, the function is copied into the isolate. The function is invoked—immediately returning a future. When function execution completes and the result has been returned, the temporary isolate is discarded. For example, `+/{⎕DL 1+⍵}⍎¨2 3 4` will create three isolates and execute three parallel delays of 3, 4, and 5 seconds respectively. The leftmost function `+/` needs to know the value of the results in order to compute the sum, and will block until all the delays have completed. The expression can be expected to complete in roughly 5 seconds, returning a floating-point result slightly in excess of 12 (the sum of the time waited by the three parallel delays).

In addition to the temporary isolates created by ⍎, the user can explicitly create isolates and populate them with functions and data that become available for repeated asynchronous use. For example:

```
megaroll←?4 1e6⍴6      ⍝ Simulate 4 million dice throws
row←⍪¨4⍴⊂⍬             ⍝ Create 4 empty isolates
row.rolls←↓megaroll    ⍝ Split into 4 rows; transfer each to an isolate
avg←row.((+⌿÷≢)rolls)  ⍝ Start computing the 4 averages in parallel
(⌈⌿,⌊⌿)avg             ⍝ Compute the highest and lowest average
```

The penultimate expression, which starts the four parallel average computations, immediately returns four futures without waiting for the computations to complete. The last line blocks when the first function which requires the values (`⌊⌿`) is invoked, until all four averages are computed.

Futures and isolates were designed to provide a tool for deterministic parallel and asynchronous programming, in the sense that the *parallel* operator ⍎ can be applied to any function within a body of code, without changing the meaning of the code (so long as the function in question has no side effects). For a language which is at its core an executable mathematical notation, the ability to provide hints about sections of code which are likely to be worthwhile to execute in parallel is a simple model for parallel programming.

Futures and isolates are inspired by a much older language feature found in some of the earliest APL systems. Since APLSV in 1973 [Falkoff and Iverson 1973a], the use of shared variables allows APL to communicate asynchronously with service processes such as full screen managers and file system interfaces. Many APL systems extended the mechanism to allow sharing variables between two APL sessions. IPSA eventually added a "network shared variable processor" [Potyok 1987] that allows APL sessions running on different computers to share variables, transfer information, and implement global trading systems and electronic mail applications.

Shared variables provide a simple mechanism for asynchronous and thus also parallel processing. Shared variables allow each end of a conversation to decide whether to operate in a synchronous or asynchronous mode. In synchronous mode all an APL program needs to do to request an external service is to set a variable to a value containing a command; any reference to the variable simply blocks until the external process sets the variable to contain a response (which can be an APL array of any rank, shape, and type). If the client program does not want to be blocked waiting for the response, it can inspect the state of the variable and continue doing other work until the response arrives.

Futures allow each item of an array to be manufactured asynchronously. In current Dyalog APL, the only way to create a future is to execute an expression within an isolate. However, there is no reason why the mechanism cannot be extended to allow a multi-threaded application to explicitly create futures to produce array items, that will block any requests for the items until the value is produced.

Futures have also inspired a proposal for a closely related *lazy* (or, whimsically, *Schrödinger*) operator, denoted ⍰ in the example below. The *lazy* operator would be just like the *parallel* operator, except that it would return a lazy future. Lazy futures are similar to eager ones except that execution of the operand function is deferred until a reference is made to the item containing the result:

```
cat←{(×?10)⊃'dead' 'alive'}   ⍝ 9x as likely to be alive as dead
cats←cat⍰¨10⍴0                ⍝ An array of 10 lazy cats
cats[2]                       ⍝ Invoke the function "cat" once to
                              ⍝   determine the well-being of cat #2
```

Futures originated in the parallel processing and AI communities [Friedman and Wise 1976; Baker and Hewitt 1977].

4.5 Sparse Arrays

The dialect J provides support for sparse arrays through a single primitive function *sparse* [Hui 1998], denoted here by `$` to avoid confusion that would be caused by a mix of APL and J notation (it is denoted by `$.` in J). The sparse representation does not store "zeros" (the sparse element): internally, there is an array of non-"zero" cells, a corresponding array of indices for those cells, and a sparse element (and other things). The sparse function converts a dense array to a sparse equivalent; it is the identity function on a sparse array; and its inverse converts a sparse array to a dense one.

```
   d                    $ d                  2 + $ d              $⍣¯1 ⊢2 + $d
 0 75  0 53          0 1 │ 75          0 1 │ 77           2 77  2 55
 0  0 67 67          0 3 │ 53          0 3 │ 55           2  2 69 69
93 51 83  0          1 2 │ 67          1 2 │ 69          95 53 85  2
                     1 3 │ 67          1 3 │ 69
                     2 0 │ 93          2 0 │ 95
                     2 1 │ 51          2 1 │ 53
                     2 2 │ 83          2 2 │ 85
```

The *sparse* function `$` also has a dyadic case which (among other things) allows a sparse array to be created without first creating a dense equivalent. For example, `1$(⊂s),(⊂a),⊂e` is a sparse array with shape `s`, sparse axes `a`, and sparse element `e`. (The non-"zero" elements of such an array can then be populated by use of the J equivalent of the *at* operator `@` (§3.7).)

The guiding principle is the identity `x ≡ $x`, from which is derived the identities:

```
f  ↔  f⍤$
f  ↔  f⍤($⍣¯1)
```

for any array `x` and for any function `f`, with the possible exception of those (like *overtake* `↑`) which use the sparse element as the fill. (Reminder: `≡` is the APL function *match* and ↔ is a metalinguistic symbol denoting equivalence.)

For example, if `x` is a sparse array with a sparse element of `0`, then the sparse element of `*x` (e^x) is `1`; and if `y` is a sparse array then the sparse element of `x+y` is the sum of their respective sparse elements. Letting the sparse element be variable rather than fixed at zero makes many more functions closed on sparse arrays (e.g. `*x` or `2+x`), and familiar results can be produced by familiar phrases (e.g. `⌊0.5+x` for rounding to the nearest integer).

Some amusing effects are possible: If `x` is sparse then `sort x` is also sparse but `⍋x` is dense and may fail due to insufficient memory. It is possible for `×≠⍴x`, the product of the shape of `x`, the number of elements in `x`, to be `∞`.

Sparse arrays provide another example of "overloading"—there is not another set of functions for operations on sparse arrays, either derived from an operator (`+S`, `-S`, `×S`, etc.) or a different set of related symbols (+̲, -̲, ×̲, etc.). For example, to find the sum of two arrays, all four combinations of dense and sparse, you just say `x+y`; to find the sum of a vector, dense or sparse, you write `+⌿x`.

4.6 Infinite Arrays

Infinity and infinite arrays were specified in 1978-1981 [Iverson 1978b; McDonnell and Shallit 1980; Shallit 1981]. Although they are not yet implemented, meditation on infinite arrays sharpens and expands our understanding of arrays. As described above (§4), an array is a function from a set of indices onto some domain. APL rectangular arrays to-date have been implemented by enumerating the array elements in row-major order (and employing the "implementation trick" of not storing the indices). But there are ways to represent a function other than enumerating the domain and/or range of the function.

Infinite arrays can be implemented by specifying the index function. For example, the index function for `⍳∞` is the identity function, `⊢` or `{⍵}`. There is a precedent for such an infinite vector/function in APL. In J, the function `p: n` is the `n`-th prime (`p:` ↔ π⍣¯1, the inverse of the prime counting function $\pi(n)$.)

Let `x` and `y` be infinite vectors with index functions `fx` and `fy`. If `s1` is a scalar monadic function, then `s1 x` is an infinite vector and its index function is `s1∘fx` or `{s1 fx ⍵}`, `s1` composed with `fx`. If `s2` is a scalar dyadic function, then `x s2 y` is an infinite vector and its index function is the train `fx s2 fy` or the dfn `{(fx ⍵) s2 (fy ⍵)}`.

In the following examples, the infinite vectors are anotated with a corresponding index function, both as a tacit expression and as a dfn. Some obvious simplifications have been applied.

```
   ⍳∞                                  │
0 1 2 3 4 5 6 7 8 9 10 11 12 …         │  ⊢              {⍵}
                                       │
   ∞ ⍴ 2                               │
2 2 2 2 2 2 2 2 2 2 2 2 2 …            │  ⊢∘2            {2}
                                       │
   - ⍳∞                                │
0 ¯1 ¯2 ¯3 ¯4 ¯5 ¯6 ¯7 ¯8 ¯9 …         │  -              {-⍵}
                                       │
   ⊢ x←3*⍳∞                            │
1 3 9 27 81 243 729 2187 6561 …        │  3∘*            {3*⍵}
                                       │
   ⊢ y←(⍳∞)*2                          │
0 1 4 9 16 25 36 49 64 81 100 …        │  *∘2            {⍵*2}
                                       │
   x+y                                 │
1 4 13 36 97 268 765 2236 …            │  3∘* + *∘2      {(3*⍵)+(⍵*2)}
                                       │
   ⍟ ⍳∞                                │
DOMAIN ERROR                           │
                                       │
   !⍳∞                                 │
1 1 2 6 24 120 720 5040 40320 …        │  !              {!⍵}
                                       │
```

```
   3*⍳∞                                  │
1 3 9 27 81 243 729 2187 6561 …          │  3∘*              {3*⍵}
                                         │
   x ≡ 3*⍳∞                              │
1                                        │
                                         │
   (3*⍳∞) ÷ !⍳∞                          │
1 3 4.5 4.5 3.375 2.025 1.0125 …         │  3∘* ÷ !          {(3*⍵)÷(!⍵)}
                                         │
   +⌿ (3*⍳∞) ÷ !⍳∞                       │
20.0855                                  │
                                         │
   * 3                                   │
20.0855                                  │
```

5 SYNTAX

5.1 Parsing an APL Expression

We consider the parsing of an APL expression. Parsing is preceded by tokenization, and execution takes place as part of parsing. The formal parsing rules are presented as a parser model in Appendix C, adapted from *A Dictionary of APL* [Iverson 1987, §I and Table 2] and from J [Hui and Iverson 2004, §II.E; Hui 1992a, §1.2; Hui 1993]. Alternative descriptions of parsing can be found in the reference manuals for APL2 [IBM 1994, §3] and Dyalog APL [Dyalog 2018b, pp.24-25]. For an apologia of the APL order of evaluation, see [Iverson 1966, Appendix a].

The expression to be parsed is tokenized and put into a queue, prefaced by a marker `⋄`, together with the corresponding class letters, `A` for array, `F` for function, `M` for monadic operator, `D` for dyadic operator, etc. As parsing proceeds from right to left (or bottom to top in a transposed view), tokens are pushed onto a stack. The stack is initialized with four marker entries `⋄`, which usually are not shown (and the stack is considered "empty" if it has just the four marker entries).

Parsing is controlled by the parse tables, three matrices `Cases`, `Actions`, and `Masks` (Appendix C). In particular, `Cases` is a 4-column matrix of enclosed class letters.

At each parse step, class letters of the four entries at the top of the stack are tested against successive rows of `Cases`. The applicable row (if any) of `Cases` is the first one in which all four tests are true. Call this row `i`. `Actions[i;]` is applied to the eligible portion of the four entries at the top of the stack, with eligibility indicated by `Masks[i;]`, and the result replaces the eligible portion. If no applicable row is found, the rightmost (bottom) entry of the queue is moved to the left (top) of the stack. Parsing terminates when no applicable row is found and the queue is empty.

We demonstrate the process by using the parser model on the expression `+.×⌿3×1 2`. In the simplest case, `parse` applies to a character vector of an APL expression and produces a fully-parenthesized equivalent expression or a nested array of tokens:

```
      parse '+.×⌿ 3 × 1 2'
(((+.×)⌿)(3×1 2))
```

```
      1 parse '+.×⌿ 3 × 1 2'
┌─────────────┬─────────┐
│┌─────┬─┐    │┌─┬─┬─┬─┐│
│└─┬─┬─┘⌿│    ││3│×│1 2│
│  │+│.│×│    │└─┴─┴─┴─┘│
│  └─┴─┴─┘    │         │
└─────────────┴─────────┘
```

Executing `parse` with a left argument of `0 2` specifies tracing, showing the parse steps:

```
  0 2 parse '+.×⌿ 3 × 1 2'
┌─┬─┬─┬─┬─┬─┬─┬─┬─┐
│⋄│+│.│×│⌿│3│×│1 2│
├─┼─┼─┼─┼─┼─┼─┼─┴─┤
│⋄│F│D│F│M│A│F│A  │
└─┴─┴─┴─┴─┴─┴─┴─┴─┘
```

`Move`

The first parse step. The queue has the tokenized original expression in row 0 and corresponding class letters in row 1. The stack is empty. No row of `Cases` is applicable, so a move is specified.

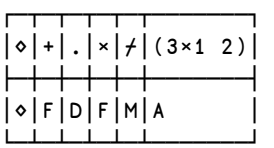
`...` 3 more move steps

```
┌─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┐
│⋄│+│.│×│ │⌿│3│×│1 2│
├─┼─┼─┼─┤ ├─┼─┼─┼─┴─┤
│⋄│F│D│F│ │M│A│F│A  │
└─┴─┴─┴─┘ └─┴─┴─┴─┴─┘
```

`2 Fn2`

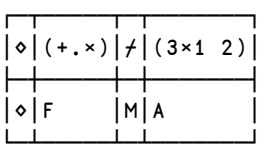
After 4 moves the queue has 4 fewer entries and the stack has 4 entries.
The top (leftmost) 4 entries of the stack `M A F A` test true on row 2 of `Cases`: the dyadic function `×` is evaluated, producing `(3×1 2)`.

`...` 4 move steps



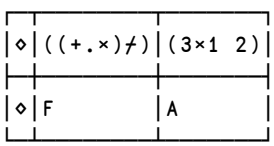
4 Op2

After 4 more moves the queue is empty.
The top (leftmost) 4 entries of the stack ⋄ F D F test true on row 4 of Cases: the dyadic operator . is evaluated, producing (+.×).



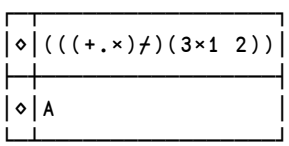
3 Op1

The top (leftmost) 4 entries of the stack ⋄ F M A test true on row 3 of Cases: the monadic operator ≠ is evaluated, producing ((+.×)≠).



0 Fn1

The top (leftmost) 4 entries of the stack ⋄ F A ⋄ test true on row 0 of Cases: the monadic function ((+.×)≠) is evaluated, producing (((+.×)≠)(3×1 2)).



The top (leftmost) 4 entries of the stack ⋄ A ⋄ ⋄ does not test true on any row of Cases, and the queue is empty. Parsing terminates.

(((+.×)≠)(3×1 2))

The result of parse.

In parse, no expression in the argument is actually executed; instead, eligible expressions are catenated together into a longer expression. An actual parser in an APL interpreter *would* execute.

Common cases of the rules embodied by the parse tables Cases, Actions, and Masks are summarized as follows. It is often easier to parse with these summary rules, and use the formal rules only to resolve more difficult cases.

- Operators are executed before functions; for an array a, the phrase ,⍤2-a is equivalent to (,⍤2)-a, not to ,⍤(2-a). Moreover, the left argument of an operator is the entire function phrase that precedes it. Thus, in the phrase +.×≠a, the operator ≠ applies to the function derived from the phrase +.×, not to the function ×.
- A function is applied dyadically if possible; that is, if preceded by an array that is not itself the right operand of an operator.
- In the bracket-semicolon indexing, a[i] or a[i;j;...], the expressions (if any) between the brackets and separated by semicolons (if any), are executed from right to left. The resultant "index expression" is used to index into the array on the left of the brackets.
- A sequence of two or more functions juxtaposed in isolation form a train (§1.3). They are parsed three at a time from right to left, resulting in a new function and reducing the sequence length by two. A sequence of even length will form a final train of length 2. The leftmost of each triplet may also be an array and is treated as a constant function.
  The isolation of the sequence can be effected by parentheses, or by having ← on its left and nothing on its right, or by the sequence being the entire expression.
- There is no order of precedence for functions.

Here, we do not consider anything which is not part of an APL expression: control structures, ⋄, comments, etc. An APL expression within a dfn (§1.2) is within our purview, but the dfn itself is parsed as a single token, as an opaque function, monadic operator, or dyadic operator. A strand is a sequence of two or more arrays juxtaposed, forming a vector whose items are those arrays; in this

paper, non-constant strands are not used (except as an example in §0.3.2) and we do not consider how strands are parsed. Finally, we do not consider how to parse some anomalous constructs, which in any case are not used in this paper.

The algorithm described above was also used for translation to a tacit expression in J [Hui et al. 1991] and for conversion from Dyalog APL to C# [Smith 2004; §8.4].

5.2 Parentheses

The APL order of evaluation tends to reduce the need for parentheses. There is a reason other than making expressions shorter or the APL designers somehow having a hatred for parentheses. Parentheses, especially parentheses with long spans, require a switch in one's train of thought and consume mental stack space. Consider the quadratic formula rendered in APL in two different ways (where `sqrt←⋆∘0.5`):

```
((-b) (+,-) sqrt (b⋆2) - 4×a×c) ÷ 2×a
(2×a) ÷⍨ (-b) (+,-) sqrt (b⋆2) - 4×a×c
```

The second expression, without the long-span parentheses, is easier to read, especially if you recognize `÷⍨` (*divide commute*) as *into* or *divided into*. In conventional mathematical notation, the quadratic formula avoids the long span parentheses by the rule that *over* has lower precedence than the other functions: $-b \pm \sqrt{b^2 - 4ac}$, *over* $2a$, or by other more complicated rules.

5.3 Function and Operator Valence

A function is ambivalent, having valence 1 when called monadically, with one array argument, or valence 2 when called dyadically, with two array arguments. (In the beginning, in *APL\360*, there were niladic functions (valence 0), long ago recognized as unfortunate design; today, their use is not encouraged.) An operator has a fixed valence of 1 or 2, whose operand can be an array or a function. Therefore, it is possible to define a "function" with 3 or 4 array arguments, by defining a dyadic operator which takes two array operands, or a "function" with 2 or 3 array arguments, by defining a monadic operator which takes one array operand. The *hypergeometric* operator in J provides an example of this.

In conventional mathematical notation, the syntax of the hypergeometric function is as follows [Abramowitz and Stegun 1964, Ch. 15; Graham et al. 1989, §5.5]:

$$F\left(\begin{matrix}a_0,\ldots,a_{m-1}\\ b_0,\ldots,b_{n-1}\end{matrix}\middle| z\right) \quad \text{or} \quad F\left(a_0, \ldots, a_{m-1}; b_0, \ldots, b_{n-1}; z\right)$$

There are `m+n` parameters a_i and b_j and a complex argument z. Ostensibly, the hypergeometric function has `m+n+1` arguments. In J, this becomes

```
⍺ (a H b) ⍵
```

where `H` is a dyadic operator (`H` is used here to avoid a mix of J and APL notation), `a` is an `m`-element vector and `b` is an `n`-element vector. `(a H b)` derives a function with ranks 0 (that is, a scalar function), where `⍵` is an array of complex numbers and `⍺` is the number of terms to sum. In the monadic case the sum is evaluated to its limit; that is, the monadic case is defined to be `∞∘(a H b)`, infinity curried to `(a H b)`.

What about your quotidian function with `n` arguments where the arguments don't necessarily have nice relationships with each other? Such a function can be defined as a monadic function; the `n` arguments can be presented as an `n`-item vector; and the first action of the function can be a "strand assignment" which assigns names to the `n` items of the argument. For example:

```
sqrt   ← ⋆∘0.5
qroots ← {(a b c)←⍵ ⋄ (2×a) ÷⍨ (-b) (+,-) sqrt (b⋆2) - 4×a×c}
```

```
   qroots 3 4 5
¯0.666667J1.10554 ¯0.666667J¯1.10554
   qroots 1 2 3 4 5    ⍝ argument must have 3 items
LENGTH ERROR
qroots[0] qroots←{(a b c)←⍵ ⋄ (2×a)÷⍨(-b)(+,-)sqrt(b*2)-4×a×c}
                           ^
```

Strand assignment was available in Iverson notation *circa* 1964 (although it was not then called strand assignment, or called anything) [Falkoff et al. 1964, cover & p.258b; Falkoff and Iverson 1978, Frame 4], but was reluctantly abandoned in *APL*360 [Falkoff 1969]. Strand assignment has made its way back in second-generation APLs.

Falkoff proposed using bracket-semicolon notation for functions with `n` arguments [Falkoff 1982], but the idea was not then taken up. k identifies bracket-semicolon indexing `A[i;j;k;p;q]` with function calls `F[x;y;z;a;b]`, and so allows functions with `n` arguments. The notation permits currying of one or more arguments, e.g. `F[;y;;a;]`.

Functions defined using the operator, strand assignment, or bracket-semicolon methods are not first-class in that operators such as *rank* or *reduce* don't work on them in general. For example, for the *hypergeometric* operator `⍺ (a H b) ⍵` the operands `a` and `b` are opaque to the *rank* operator; that is, in `⍺ (a H b)⍤r ⊢⍵` the ranks `r` apply to `⍺` and `⍵`, not to `a` and `b`.

5.4 Left *v* Right Argument

The interpretation of the left and right arguments of a dyadic function follows firmly-established custom if there is one (e.g. `-` `÷`). If the function is commutative, which is the left or right argument does not matter (e.g. `⌈` *max* or `⌊` *min*). What about a non-commutative function without an existing interpretation for the arguments? [Iverson 1962, p.265] offered that "controlling variables appear to the left". More generally, as already noted (§2.2):

> Moreover, the order of the arguments in many primitive dyadic functions, when not constrained by firmly-established custom, is such that `a∘f` (currying a left argument) is a sensible monadic function.

Awkwardness in a function having an infelicitous argument order is ameliorated by use of the *commute* operator `⍨`, where `⍺ f⍨ ⍵ ↔ ⍵ f ⍺`.

5.5 Numerals

As in other languages, APL defines a notation for numerals: `3e4` means `3` times `10` to the power `4` and `¯17.4e¯6` means `¯17.4` times `10` to the power `¯6`. A complex constant is written as the real part, the letter `j`, and the imaginary part: `¯1.2e¯3j¯45` is the complex number with real part `¯1.2e¯3` and imaginary part `¯45`. APL2 also has polar notation for complex numbers: `3.4r5.6` means a number with magnitude `3.4` and angle `5.6` radians; `3.4d¯30` means a number with magnitude `3.4` and angle `¯30` degrees.

J [Hui and Iverson 2004] and NARS2000 [Smith 2020] extend these ideas further, including:

`123456x`	`123456` as an extended precision integer
`3r7`	the (exact) rational number with numerator `3` and denominator `7`
`3.4ar2.6`	the number with magnitude `3.4` and angle `2.6` radians (J only)
`3.4ad¯30`	the number with magnitude `3.4` and angle `¯30` degrees (J only)
`2i3j4k5`	quaternion (NARS2000 only)
`2i3j4k5l6il7jl8kl9`	octonion (NARS2000 only)

6 SYMBOLS

6.1 Overview

How APL designers consider notation and symbols is exemplified by two quotations presented at the beginning of *Notation as a Tool of Thought* [Iverson 1980]:

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.
>
> — A.N. Whitehead

> The quantity of meaning compressed into small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid.
>
> — Charles Babbage

From the beginning APL was designed for communication among people more than for communicating with automata, executable mathematical notation rather than a programming language. This has influenced its use of symbols to this day.

When *APL*\360 was implemented, the alphabet (character set) then available on computers was extremely limited. A standard terminal for System/360 was the IBM 2741, which is based on the IBM Selectric™ typewriter with a changeable typing element ("typeball"). The *APL*\360 team exploited the technology and designed a typing element with a new 88-character alphabet [Iverson 1991a]. Many of the characters were special to APL.

Nowadays, the APL special characters are "no longer a problem"; at least, they are no longer a problem if there is an implementer with the knowledge and skills to solve the problems. Other ways the special characters are no longer a problem are the common availability of richer (ASCII) and much richer (Unicode) alphabets and programmable devices for input and output. Some dialects (J, k) use ASCII, possibly with digraph spelling; other dialects (Dyalog APL) use the Unicode encoding of the special APL characters, which are no longer so special in a sea of over 100,000 Unicode symbols.

Choosing a symbol for new functionality is straightforward if the Unicode code point and glyph already exist. In Dyalog APL, a heuristic is to examine the APL section of Unicode, less the characters supported by classic APL:

```
      ⎕av ~⍨ ⎕ucs 9014+⍳60
⌸⌺⌻⌼⌾⍁⍂⍃⍄⍅⍆⍇⍈⍊⍌⍍⍏⍐⍑⍓⍔⍖⍗⍘⍚⍛⍜⍠⍡⍢⍤⍥⍦⍧⍩⍭⍮⍯⍰
```

Two of the above have recently been taken into use: ⌸ U+2338 for *key* (§3.3) and ⌺ U+233A for *stencil* (§3.4). If a desired glyph does not yet exist then its creation is, in the words of Iverson, "not impossible but requires an act of Congress" [Iverson 2004].

The rest of this section traces the evolution of APL symbology, the problems encountered and overcome, the various methods devised to address the limitations of standards and technology extant, etc.

6.2 Vailing and Countervailing Pressures

Pressure to introduce new symbols is reduced by several factors. The first five (0–4) were known and widely exploited by 1978:

0. Ambivalence. A function (nearly always) has a monadic definition and a dyadic definition. For example, monadic `-` is *negate* and dyadic `-` is *minus*.

1. Overloading. A single symbol `+` for adding integers, floating-point numbers, complex numbers, rational numbers, sparse arrays; a single symbol `,` for appending numbers, characters, nested/boxed arrays, dense/sparse arrays; and so on.

2. Encoding with one argument. The technique is exemplified by the primitive scalar dyadic function *circle*, where `1○⍵` is *sine*, `2○⍵` is *cosine*, `3○⍵` is *tangent*, ..., and `¯1○⍵` is *arcsine*, `¯2○⍵` is *arccosine*, etc. [McDonnell 1977]. The potential and pitfalls of the technique were recognized long ago. From *The Design of APL* [Falkoff and Iverson 1973b]:

   > The use of the circle to denote the whole family of functions related to the circular functions is a practical technique for conserving symbols as well as a useful generalization.
   > ...
   > The notational scheme employed for the circular functions must clearly be used with discretion; it could be used to replace all monadic functions by a single dyadic function with an integer left argument to encode each monadic function.

   The potential was further exploited when complex numbers were introduced, where new functions were defined by new cases of *circle* without requiring use of new symbols: `9○⍵` is *real part*, `10○⍵` is *magnitude*, `11○⍵` is *imaginary part*, `12○⍵` is *phase*, etc. [Penfield 1979; Penfield 1981; McDonnell 1981a]. On the other hand, it was consciously avoided when Dyalog APL added the *stencil* operator (§3.4). A "cut" operator already existed in SHARP APL and J which used an integer operand to select one of several alternative cutting operations. Dyalog APL elected to use a new symbol `⌺` (visually suggesting a smoothing operation applied to a window) to provide a subset of the functionality available through *cut*.
3. An operator provides a family of functions with just one symbol. Thus, for example, there is no need to introduce Σ or Π or ⋀ or a symbol for "power tower", because the computations derive as `+⍟` and `×⍟` and `∧⍟` and `*⍟`.
4. The so-called `⎕` names (quad names) were introduced with APLSV in 1973 [Falkoff and Iverson 1973a] in the form of "system" variables and functions (`⎕io`, `⎕ct`, `⎕fx`, `⎕nl`, `⎕svo`, etc.), and file system functions were introduced in SHARP APL and STSC APL at about the same time (`⎕fread`, `⎕freplace`, `⎕fappend`, etc.). A dichotomy emerged of functions and operators in "core language", using symbols, and "system functions", for access to the APL environment, file systems, GUI, external facilities, and so on, using `⎕` names.

The last three (5–7) were realized only later:

5. In second-generation APLs user-defined functions are first-class, that is, can be used as operands to operators. It is no longer necessary to make a function primitive, and use a symbol to denote it, in order for an operator to apply to it.
6. Operators expand the encoding possibilities (§3). A monadic operator (a single symbol) can embody four different families of functions, and a dyadic operator (again a single symbol) can embody eight different families of functions.
7. The emphasis on the leading axis, together with the *rank* operator, eliminates the need to have two symbols for a function, one for the leading axis and one for the trailing axis (§2). For example, `f⍟` and `f⍟⍤1` (instead of `f⍟` and `f/`) provide for reduction on the leading axis and on the trailing axis, respectively.

Additionally, there are non-notational factors with damping effects on introducing new symbols. A new symbol requires a keyboard sequence for its entry, changes in keyboard maps (software or printed), an element in a font for its display, and changes in the documentation describing same. If backward compatibility (§10.3) is a requirement, one also needs to consider whether there may be a better use of the new symbol, because once introduced the usage is forever and can not be changed. (Paradoxically, backward compatibility also induces pressure *for* introducing new symbols. See below.)

On the other hand, there are only two sources of pressure for introducing new symbols.

The first is that every new primitive function or operator needs a new symbol. Recent examples arose in the work on the total array ordering [§2.5; Brudzewsky et al. 2018] where functions are needed to discuss the ordering between two arrays. In conventional notation,

`⍺ ≼ ⍵` `1` iff `⍺` precedes or equals `⍵`
`⍺ ≽ ⍵` `1` iff `⍺` follows or equals `⍵`
`⍺ cmp ⍵` `¯1, 0,` or `1` according to whether `⍺` precedes, equals, or follows `⍵`

One imagines that in another programming language these functions would be (could be) introduced without much fuss. Brudzewsky *et al.* never considered making these functions primitive.

Another countervailing current, one imposing more constraints, is the requirement for backward compatibility (§10.3). The designer is not free to use the "perfect" symbol for a new function because that symbol already has a meaning inconsistent with the new, preferred understanding, or is already used for something else. Two examples are *member* (`∈`) and *tally* (`≢`).

Member is denoted in mathematics and in APL as `∈`. As described in §2.2, one would like to extend `∈` to look for major cells. Unfortunately, in current APL `⍺∈⍵ ↔ ⍺∈,⍵`, as though the right argument were ravelled (first made into a vector), and `∈` can not be extended and remain backward compatible. Instead one is forced to consider using `∋`.

The symbol `∈` also provides a riposte to the claim that special symbols are "no longer a problem", in two different ways: First, the use of `∈` (U+220A) as an APL primitive poses problems for Greek speakers who wishes to use `∈` (U+03B5) in user names. Second, as of late February 2020, Adobe Acrobat Reader DC (20.006.20034) does not correctly handle `∈` in a PDF file. Copy and paste of `∈` from APL code changes it into the Greek `∈` and would fail to execute in an APL session. (How many other software packages have this problem?)

The second example of contrained symbology is the function *tally*, denoted `≢` (§2.1). The number sign `#` would have been just the symbol; moreover, the dyadic meaning could have been *replicate*:

```
      # 'Empyrean'                    4 3 2 1 0 1 2 3 # 'Empyrean'
8                                  EEEEmmmppyeaannn
```

These are the definitions of `#` in J. Unfortunately, by the time *tally* was introduced in Dyalog APL, `#` had long been used to denote the root namespace (§4.3). One can wish wistfully but uselessly that U+2302 ("house" or "home plate") had been available and used for that purpose instead. NARS2000 uses `≢` for *tally* for similar reasons [Smith 2019].

6.3 Composing Symbols

6.3.1 Overstriking

The first APL terminal was based on the IBM Selectric™ typewriter, and the first APL 88-character alphabet was designed so that new characters can be formed by "overstriking", for example, `⌸` is formed by typing `⍞`, backspace, `÷`. In general, the rule is "visual fidelity": on a line of input, any combination formed by space, backspace, tab, and the 88-character alphabet is accepted, as long as it "looks like" one of the valid characters.

An informal study was done to determine (by eye) which pairs of overstruck characters were suitable as new symbols [Penfield 1975]. Forty possibilities were identified. The study missed a few, but the number of serviceable overstrikes were not many more than that. Therefore, the APL character set was limited to about 128 (88+40) even with overstriking.

In any case, the usefulness of overstriking as a technique was soon coming to an end: dot matrix impact printer terminals were coming into widespread use and the lower output resolution compared to that of "typeball" or "daisy wheel" technologies further reduced the number of legible overstrikes. Then, video display terminals made it so that new overstruck combinations could be supported only with difficulty (at least for the early "dumb" terminals).

Overstriking echoes with an amusing counterpoint from the early days. When it came time to typeset *A Programming Language* [Iverson 1962], no sorts for ⌊ (*floor*) and ⌈ (*ceiling*) were available, and new sorts had to be created by filing away a "serif" of the sort for [[Iverson 2004; Graham et al. 1989, p.67]. (These are probably not the "sort" and "file" you expect to see in this paper.)

6.3.2 Multicharacter Symbols

In J, symbols are spelled with either one or two character from the 7-bit ASCII alphabet, with the second being a period or colon. (The rules permit more than one trailing . or :, but with two exceptions are not exploited.) For example,

+	*conjugate*	•	*plus*	%	*reciprocal*	•	*divide*
+.	*real/imag*	•	*GCD (or)*	%.	*matrix inverse*	•	*matrix divide*
+:	*double*	•	*not-or*	%:	*square root*	•	*root*

The J spelling scheme does not offer the opportunity—nor imposes the burden—of designing a suitable glyph for a symbol, but is nevertheless symbolic and mnemonic.

Dyalog APL has three examples of multicharacter, non-overstruck symbols: the ⍺⍺ ⍵⍵ ∇∇ used in dfns to define operators. At the time dfns were introduced (1996), there were no Unicode APL implementations and the "obvious" alternatives ⍺̲ ⍵̲ ∇̲ were not in the Dyalog APL character set then extant [Scholes 2018].

6.3.3 Single ASCII Symbols

k went a different way, eschewing *both* the special characters and multicharacter symbols, and uses single 7-bit ASCII characters to denote primitives. All primitives were considered with a critical eye; only the most crucial functions and operators were deemed worthy of being a primitive.

The severe constraint imposed by the single ASCII character symbols does lead to a useful language design heuristic: if a function is a primitive in k, consuming Whitney's most precious resource, it must be worthy of careful study and consideration. For example, # is *tally* in k (§2.1).

6.3.4 Unicode

Widespread support for Unicode has created many new opportunities. Not only does Unicode contain all APL symbols that are in use in current APL systems, it also includes all the "suitable" overstrikes that were discovered when the original APL typeball was designed. Examples of recent use include ⌸ *key* and ⌺ *stencil*. Unicode also contains a wide variety of mathematical or logical symbols that are candidates for use as APL primitives. For example, the root symbol √ contemplated for APL years ago [McDonnell 1986] is now available.

In addition to making symbols readily available, ubiquitous Unicode support has allowed APL users and vendors to make keyboarding mechanisms available in most popular environments. Linux distributions released after mid-2012 include keyboard support for APL as a standard xkb overlay. Under Microsoft Windows, the Dyalog "Input Mode Editor" or IME allows the entry of APL characters into any Windows application. Similar support is available under macOS using `.keylayout` files.

Finally, the use of Javascript as an implementation language for portable application user interfaces has allowed the addition of APL keyboard support to systems like tryapl (http://tryapl.org) and Electron-based editors like Microsoft VS Code and the Dyalog Remote IDE [Dyalog 2019].

See §0.4.4 for other effects of Unicode on APL systems.

6.4 Exercises

One time, Hui teased Iverson[*citation needed*] that the encoding possibilities …

of the multi-character symbols of the J spelling scheme,
of operators,
of complex numbers, and
of boxed arrays,

provided such an embarrassment of riches that *even he* would be hard put to "fill up all the cases".

No, Iverson and colleagues did not manage to fill up all the cases, and to date there are only 136 symbols in J, in addition to the space, 0–9, A–Z, and a–z. It would be an interesting exercise to design Unicode code points and glyphs for J symbols not yet in APL. (You wouldn't necessarily want to add all those primitives to APL, but that is not the point of the exercise.) Unicode has up to `2*21` code points, so how hard can it be?

To make things easier and more concrete, instead of 136 symbols, consider just three. The first example is the *sparse* function described in §4.5. It is not yet in APL. If it were to be introduced to APL, what symbol would you use? The temporary symbol `$` suggested earlier is a good candidate. `$` is in the 7-bit ASCII alphabet, prime real estate, so before it is "used up" you need to rule out that there is no other primitive even more worthy of the symbol, and if your APL dialect requires backward compatibility you need to rule out other contenders forever.

The second example is the function described in §2.6, defined by `{⍺←0 ⋄ ⍺+⍵×0j1}¨` and denoted by the 2-character symbol `j.` in J. In conventional mathematical notation, the monad is $\omega \times \sqrt{-1}$ or $i\omega$, and the dyad is $\alpha + \omega \times \sqrt{-1}$ or $\alpha + i\omega$. What symbol would you use?

The third example is the *hypergeometric* operator described in §5.3. There, the temporary symbol used is `H`, which is not available because it conflicts with user names. Designing a good symbol for it requires understanding what the hypergeometric function does and why it's so named: Should the emphasis be on "hyper" or "geometric"? Should it remind of its definition or of its use? Should it commemorate its inventor? Etc.

6.5 Names

With the challenges of using symbols, one might ask whether it may be better to use names as in most other programming languages. In fact, APL already uses names—the "system functions" mentioned above (§6.2, #4), `⎕fread`, `⎕svo`, `⎕nl`, etc.—for access to external and environmental facilities. The `⎕` names create a set of reserved words, and the `⎕` prefix makes them easily distinguishable from user names (which are not allowed to have the `⎕` character). Names are also used to effect control structures (§1.1).

What about using names for the "core language"? The following renders the quadratic roots expression in two ways:

```
(2×a) ÷⍨ (-b) (+,-) ⍘ (b*2) - 4×a×c
(2 times a) divide commute (minus b) (plus append minus)
   sqrt (b power 2) minus 4 times a times c
```

One imagines that the second rendering would be even less attractive if each name were prefaced by `⎕`. A designer undeterred by unattractiveness nevertheless needs to make some decisions with sensitivity towards the design of APL:

- APL primitives have well-chosen names: *ravel*, *floor*, *ceiling*, etc. (§10.1), so part of the naming problem is already solved: `ravel`, `floor`, `ceiling`. However, APL functions are ambivalent with the monadic and dyadic interpretations yoked to the same symbol: `,` is both *ravel* and *append*, `⌊` is both *floor* and *min*, `⌈` is both *ceiling* and *max*, etc., but `-` can be *minus* in both the monadic and dyadic interpretations (as in the quadratic roots expression above). Does

one use different names for the monadic and dyadic functions, and abandon the idea of ambivalent functions? Or perhaps some named functions are ambivalent and some not?

- Exactly what are named makes a difference. For example, APL has the *reduce* operator, denoted by `⌿`, so that `+⌿` is *sum*, `×⌿` is *product*, `⌈⌿` is *max*, etc. If `sum`, `product`, `max`, etc. are provided but not `reduce`, the essential idea of operators is discarded; if both the operator and the derived functions are named, awareness is reduced that *sum*, *product*, *max* are functions from the same family. With the inner product, the temptation is even stronger to name just `innerproduct`, for `+.×`, without making the operator available as `dot`.
- Names are commonly in some natural language. Choosing suitable non-English names would be a non-trivial task. (Aside: Symbols can also be problematic in this regard. Existing use of `⍺ ⍵ ∈ ⍳ ⍴` preempts Greek speakers from using them in user names.)
- How are primitives distinguished from user names? Or do you just ignore the problem, or argue that it is an advantage that they are indistinguishable?

7 USER INTERFACES

As mentioned in §0.4, APL users often have software engineering as a second or third skill set, after something like actuarial science, bond trading, or chemical engineering—sometimes combined with a management role. Since the mainframe days, APL vendors have worked hard to provide mechanisms for interfaces in general, and user interfaces in particular, which were based on the array paradigm and allowed APL users to create functional user interfaces. At the same time, tools are needed to provide sufficient access to the underlying libraries to allow sophisticated users to provide state-of-the-art user experiences.

This section covers some of the most successful APL UI tools in the last 40 years.

7.1 Textual User Interfaces

A number of different platforms provided character-based interfaces. Early video screens were divided up into 24 rows and 80 columns, sometimes providing a choice of 16 colors (or shades of green or orange) and/or a small number of attributes like underlining or blinking text. Examples of common environments supporting this style of interface include:

- IBM 3270 terminals connect to IBM mainframes
- VT100 terminals connected to UNIX machines
- Personal computers and workstations
- VT100 terminals connected to emulation programs which translated IBM 3270 screens into VT100 data streams
- PC applications emulating VT100 terminals, connected to emulators

The user typically placed a number of *fields* on the screen, and an APL matrix with one row per field was a natural way to represent this. The exact contents of this matrix varied across interpreters. In APL*PLUS/PC the column definitions were:

0. beginning row for the field
1. beginning column
2. number of rows
3. number of columns
4. type: 0=output, 1=input, 2=numeric input only
5. special feature selection
6. attribute when active (color, highlight)
7. attribute when inactive

APL allows easy generation of screen layouts using this kind of definition. For example, if you wanted to put a 10-character input field at column 3 on every other row of a 24×80 screen, you needed a matrix with starting row numbers in the first column. The next four columns would be the same for every field: 3 (the column number), 1 (# of rows), 10 (# of columns), 1 (input). In APL*PLUS/PC, that could be written:

```
⎕win (2×1+⍳12),12 4⍴ 3 1 10 1
```

AP124 on mainframes, `⎕win` under APL*PLUS/PC, `⎕sm` in Dyalog APL, and similar features of other PC and workstation APLs made it straightforward for non-technical APL developers to provide state-of-the-art character-based interfaces. Second-generation APL systems typically included the data content of fields as one of the columns in the matrix, further simplifying programming. Many of the resulting products often fit the technical content of the applications in ways that are rare in the current era of standardized interfaces. Developers and users usually loved the resulting products and APL was a competitive tool for building UI in the age of text.

7.2 Microsoft Win32

UNIX systems have provided a wide variety of GUI APIs based on the X Window system—and perhaps in the future OpenGL. Although various APL systems did try to provide hooks into these systems, the variety and state of constant flux of UNIX-based APIs made it very difficult for vendors to tightly integrate these APIs into APL systems. Popular APIs that have survived the test of time include GTK+ and Qt. J comes with an interface to Qt.

Financial institutions, which provide the bread and butter for most APL vendors, adopted Microsoft Windows quite quickly when it arrived, at least for the kind of applications that needed good user interfaces. Microsoft's Win32 API became the basis for GUI application development in the major APL implementations (APL2, Dyalog APL, APL+Win, and APLX) for a couple of decades.

The Win32 API was designed by and for C and C++ developers (the same is true for GTK+ and Qt). Although it would be possible to simply use the Foreign Function Interfaces that most APL systems have to invoke individual API calls (and some vendors did start there), this creates an environment where non-technical users accustomed to arrays with value semantics need to track memory allocations and deal with pointer structures. In a word: disaster.

Thus, the APL vendors all set off to create higher-level objects with array properties. For example, a typical ListBox has a property called Items. Instead of populating it using a series of function calls, which is what a C programmer would do, an APL developer would want to set it to a vector of character vectors, and have the interface do the required looping in C. A Grid object has a property containing data, which the APL user would want to set (and retrieve) as a single matrix, containing all the values for all the cells.

The market was fiercely competitive at the time, and no attempt was made to collaborate or standardize the APL covers for Win32—although MicroAPL did adopt APL2000's APL+Win model and used it to wrap a cross-platform API which could produce the same UI under Windows, Linux, and the Macintosh. While APL2000, IBM, and MicroAPL used objects named using character strings but held outside the APL workspace, Dyalog exploited namespaces and subsequently full OO technology to integrate the objects directly into the workspace, arguably making the resulting API significantly more powerful and easier to use.

7.3 .NET and Other Object Frameworks

The "C++ API" era described in §7.2 was a frustrating time for APL users and vendors alike. The APIs mutated rapidly, requiring of the vendors critical decisions and constant work to provide APL users with usable interfaces, to the extent that the vendors were able to keep up. APL users had little interest in APIs and just wanted their applications to keep working. Some vendors made fatal mistakes: the cross-platform API selected by MicroAPL as a basis for their UI ceased to be supported, ultimately causing MicroAPL to withdraw APLX from the market as they were unable to justify the investment required for a rewrite, at a time where user interfaces were entering another state of flux.

Modern object frameworks provide a much friendlier environment, both for users and implementers, because most APIs follow very similar patterns. Some have started to support the use of lists and arrays for parameters, although many APIs still require loops and lots of small arguments, making it difficult to use them efficiently from an interpreter.

The .NET Framework was particularly easy to provide an APL bridge to, because it supported not only *reflection* but also the dynamic *emission* of code on the fly. The significance of this was that an APL interpreter could have a completely generic bridge that could dynamically invoke any .NET class—and that .NET stubs could be generated for APL classes, making them appear to be regular .NET classes. As a result, APL users could use the same APIs for GUI development as

users of any other .NET language, and they could do so by interactively inspecting API definitions from within APL. The .NET GUI APIs used from APL included WinForms, Windows Presentation Foundation (WPF), and web pages via ASP.NET.

7.4 Data Binding

In the context of user interfaces, data binding is a technique that *binds* attributes of the user interface to data structures that can be referenced and modified by application code using standard language constructs (rather than API calls). As the name suggests, it usually refers to the data *content* of UI elements, but many systems also allow the binding of *attributes* like the color and size of UI elements to variables. In essence, variables are shared between the application and the user interface.

Several early array languages provided data binding. In Dyalog APL, the system variable `⎕sm` (screen map) is a matrix similar to the argument to the APL*PLUS/PC `⎕win` function shown in §7.1, with an additional column defining the content of each field. This matrix can be manipulated by the application to move, resize, or change the color of fields, or the data column can be modified to change the data in the field. As the user modifies the data, changes are immediately visible to the application. The individual fields are not bound to variables; the entire screen is mapped by a single array.

The most impressive example of data binding was probably the "electric" interface in A+, a precursor to k implemented by Whitney at Morgan Stanley in 1989-1993 [A+ 2003]. In A+, any variable could be mapped to the user interface simply by setting an attribute of the array to select a display class. If `sales` is a 2-dimensional array of numeric data, all that is required to cause it to appear in a grid for viewing and editing is:

```
sales has (`class;`matrix)
show `sales
```

Any changes made by the user while viewing the data are immediately reflected in the bound array. A default representation is immediately provided and can be refined by setting assigning attributes of the array. Examples of display classes include:

- simple classes for displaying individual global variables: array, hgauge, label, matrix, page, scalar, vgauge, and view
- classes for data entry entry: command, hscale, password, text, and vscale
- container classes: table and graph

The A+ GUI framework is documented in the Screen Management section of [A+ 2003].

On the .NET platform, Dyalog APL arrays can be used with WPF data binding.

7.5 APL as a Service

It is worth pointing out that the evolution of modern computing platforms is rapidly making it easier to integrate calculation engines written in APL with a very wide range of user interface tools, including alternative presentation technologies such as Jupyter notebooks, which are usable by less technical users.

Modern APL systems can be called as .NET assemblies, shared libraries, and as TCP services. In the future, it may become less critical for APL vendors to provide user interface technologies integrated with the APL interpreter, allowing them to focus on producing improved APL language engines.

8 COMPILERS

Although virtually all APL code in use today is executed by interpreters, there have been many substantial attempts to write APL compilers. A paper by Graham Driscoll and Donald Orth [Driscoll and Orth 1986] on the IBM Yorktown APL Translator, written at the tail end of the first wave of APL on mainframes, contains references to papers on many early compilers, including the Burroughs APL-700 and Hewlett-Packard APL-3000 compilers, and Timothy Budd's compiler for APL under UNIX in the PDP/11.

Of the early projects, only one was ever made generally available as a commercial product: the STSC APL Compiler implemented by Clark Wiedmann and Jim Weigang [Weigang 1985], available for APL*PLUS on the mainframe. This was a partial compiler which required the interpreter to be present at runtime. It compiled individual functions within a workspace, sometimes requiring slight changes to the code like declaration of sub-functions—and no use of nested arrays. The compiler removed the overhead that an interpreter has when executing APL operations on small arrays, and called into the interpreter for complex operations such as grading arrays.

STSC used its compiler to optimize parts of a standard library of APL functions. Roy Sykes, VP Marketing at STSC and manager of the compiler project, estimated that something like 10% of the organizations that used APL*PLUS on in-house mainframe computers also compiled some of their own code [Sykes Jr. 2019]. Unfortunately, the product generated IBM System/370 machine instructions and disappeared along with the APL*PLUS mainframe product.

8.1 Lack of Adoption of Early Compilers

For 50 years, APL compilers came and went without making a significant dent on the market. The reasons for this include:

8.1.1 Interpreted APL Is Very Fast

Typically, speed becomes an issue, and compilers become important, when you have large quantities of data to process. However, since APL interpreters operate on large arrays without using explicit loops, the fixed interpretive overhead is amortized over a large number of items. APL programs essentially become optimized byte code defining sequences of array operations—and can be almost as efficient as compiled code. APL interpreters have been highly optimized for decades, switching algorithms based on the runtime data types and the range of the data, taking advantage of vector instruction sets.

Most good APL interpreters will perform high-level optimizations of operator expressions. As a trivial example, although `∧.=` as applied to two vectors is defined as the `∧` reduction of the result of comparing *all* corresponding elements of the two arguments, no self-respecting APL interpreter will actually do more comparisons than it needs to; it will stop as soon as a comparison produces 0. Similarly, `f⍣⍤r` (`f` reduce rank `r`) shortcuts the process for the general case described in §3.1 and is much faster for that, and the `{⍺⍵}⌸` phrase used in examples in §3.3 is supported by special code. Interpreters also recognize phrases known as *idioms* (§9.3): an interpreter which sees the sequence `b/⍳n` "knows" that it is being asked to find the indices of non-zero items of the Boolean left argument, and does not start by generating an array containing the first `n` integers and selecting from that.

Some APL interpreters also make a habit of "squeezing" arrays into the smallest data type which will hold the values actually present in the array at runtime. This technique was invented in an age where it was important to make careful use of every bit of a 32K workspace. In the 1980s and 1990s it started to look like a questionable strategy, but as memory speeds started to lag significantly behind CPU speeds, it became advantageous again. APL systems manage to make do with 1-bit Booleans and 1- or 2-byte integers for a lot of arrays, and APL interpreters have a lot of special code

to use SSE, AVX, and similar vector operations on small types. One of the reasons why interpreted APL often beats hand-coded C is that any "sane" C programmer will use 4-byte integers (or maybe even 8-byte integers) without thinking much about it. If the data is held in a 1- or 2-byte integer structure, this also tells you something about the range, allowing further intelligent algorithm selection (§9.2).

8.1.2 Loss of Interactive Debugging

It is normal for an APL user to spend most of the day with a half-executed function on the stack, interactively experimenting with live data, adding newly crafted code snippets a line at a time to functions. At runtime, APL systems often work with poorly-defined or mutating data formats and need sophisticated debugging features where interactive APL is used as a debugging tool, to recover from failures without needing to restart jobs which may have run for hours. Building a compiler tool chain that preserves the ability to deal with this kind of situation effectively is quite difficult.

APL is often used in environments where the speed of development is as important as the speed of execution. Debugging compiled APL code may require trying to understand how the crashing code, generated in C or some other language which the developer has no experience with, relates to the original APL code. The C debugger will probably not provide much help, especially if clever optimizations such as loop fusion have kicked in.

Additionally, modern platforms have made it easy to combine modules written in different languages. This makes it practical to write an application using idiomatic APL and then—*if* the application is successful and critical hot spots emerge—ask a specialist to rewrite the critical sections in a compiled language using a state-of-the-art tool chain for that language.

8.2 Potential Benefits of Compilers

Although APL interpreters often do well on large quantities of data, compilers do have the potential to significantly speed up APL code where the array size handled by each APL primitive is small, for example when an algorithm is written as a loop around a code block which operates on a small section of data. In this case, the overhead of the interpreter is high, compared to the time actually spent performing the operation on data.

The above also applies when operators like *each* or *rank* are used to apply primitives or small user-defined functions to subsets of a large array—for all cases which the interpreter does not recognize and optimize. Thus, compilers have the potential to reduce the special code that APL interpreter implementers need to write in order to optimize idioms and special cases of operators.

In benchmarks, compilers demonstrate speed-ups of two or three orders of magnitude on APL code which operates on small data structures. However, existing compilers can only handle "pure" computational code, which must be free of side-effects, references to global data structures, calls to relational data bases and file systems, or other operating system calls. Unfortunately, these requirements, which most software engineers would find quite natural, are frequently at odds with the use of APL as an executable mathematical notation, to build models where the exact data requirements, and the structure of the algorithms used, often change dramatically during the development process. Architecting code for parallel execution is a challenge for experienced software engineers, and APL developers often have other primary skills.

8.3 Parallel Compilers

In the past, the potential speed-ups have not been sufficiently attractive to outweigh the disadvantages mentioned in the previous sections. In the last decade, changes to both hardware and

software are making APL compilers look more attractive, at least for applications with large data parallel sections.

- For a while, CPU core speed increased relatively faster than main-memory access speeds, with high-speed cache memory close to the CPU becoming important to high performance. After clock speeds could no longer increase, manufactures placed multiple cores on the same chip, surrounded by high-speed caches. Data flow analysis across sequences of APL primitives allows better cache utilization, and significant speed-ups may be available. For example, in an expression like `R←A+B×C`, an interpreter will first multiply all of `B` to all of `C`, create a temporary result, and then proceed to add this to `A`. A compiler, sensitive to the size of the processor cache, can compute blocked, fused loops which ensure that cache is optimally utilized.
- Massively parallel techniques like GPGPU (general-purpose computing on GPUs) have become widely available. The relative cost of memory transfers is even higher in this scenario; efficient use of GPGPU by APL demands a compiler—optimizing individual primitives on highly data parallel hardware is pointless.
- Compiler theory and practice continue to evolve, and many of the array algorithms used by interpreters are being reinvented and in some cases surpassed by modern compilers.
- A growing interest in, and understanding of, functional programming as a tool for expressing parallelizable algorithms is driving the development of APL, for example dfns (§1.2), and the technology of parallel systems in general [Šinkarovs et al. 2019].

A new generation of APL compilers, designed at the outset to target parallel hardware, is emerging. Selective and just-in-time compilers, well integrated with interpreters, have the potential to remove some of the inconvenience of early compilers. Whether any of these systems will be successful remains to be seen, but the environment is growing more favorable:

- Wai-Mee Ching's ELI compiler [Chen et al. 2017] compiles ELI into C with OpenMP annotations. ELI is a language with similarities to both APL and k.
- Robert Bernecky's APEX compiler [Bernecky 1997] compiles a subset of Dyalog APL into Single Assignment C, for which highly parallel compilers exist.
- Aaron Hsu's co-dfns compiler [Hsu 2019] compiles dfns to parallel back-ends such as ArrayFire and CUDA.

Existing APL applications have been architected to get the most of an interpreter running on existing hardware. The new compilers may not provide huge speed-ups for these, but may open APL up to new highly data-parallel applications like image processing, computational fluid dynamics, and neural networks, where APL has until today been an excellent modelling language but unsuitable at runtime due to the small, local loops required by many common algorithms.

8.4 Special Purpose Compilers

Although none of the early APL compilers were really successful, several of them were used to provide critical speed-ups to APL-based products. In some cases, speed was not the only goal. For example, Adrian Smith created an APL to C# translator [Smith 2004] to create a version of his RainPro graphics package which he could sell as a managed code library to web developers on the .NET Framework, without requiring an APL interpreter as a prerequisite. Since he only needed to compile his own code, he was able to write a restricted translator that produced C# code which was still readable by anyone who knew APL. RainPro was slightly rewritten to simplify the compiler specification. The result, known as SharpPlot, is a graphics package available as a managed .NET component compiled from C#, and—on platforms where .NET is not available—a plug-compatible

set of classes written in APL. Rendering charts is often a very loopy process, and as a result the C# version of SharpPlot is significantly faster than the APL version, for certain chart types.

Like the original STSC compiler, the APL to C# translator made use of an "array engine" which performed many array operations, rather than generating low-level C# for these operations. In addition to simplifying the translator, this also had the benefit of allowing the generated C# to be fairly readable to APL developers, greatly simplifying debugging of the generated code. The SmartArrays system, created by Jim Brown and James Wheeler after they left IBM and STSC, respectively, took this a step further [SmartArrays 1999]. SmartArrays provided a high-performance array engine encapsulated in a library designed for use within applications written in C or other languages, making it possible to write array applications in environments where the use of an APL interpreter was not an option. More recently, NGN APL [Nickolov 2013] compiles a reduced APL to JavaScript, to exploit the availability of JavaScript interpreters on a wide variety of devices.

APL applications have also generated code in other languages, then compiled and run it, while the main application remains an interpreted APL system. For example, a major oil company is known to do this with FORTRAN at runtime to connect to highly optimized linear programming libraries [Bernecky et al. 1990].

9 IMPLEMENTATION

An important factor for the initial success of APL was the excellence of the implementation, for which Larry Breed, Dick Lathwell, and Roger Moore received the ACM Grace Murray Hopper Award in 1973 "for their work in the design and implementation of *APL\360*, setting new standards in simplicity, efficiency, reliability and response time for interactive systems" [ACM 1973]. Nowadays the reliability and efficiency of the implementation are even more crucial, as they are main factors in the commercial viability and success of APL. In this section we look "under the covers" at a few implementation matters.

9.1 Array Representations

APL arrays are stored in row-major (ravelled) order—consecutive items in a row are stored in consecutive memory locations; more generally, consecutive cells, and consecutive items in a cell, are in consecutive memory locations. Equivalently, in terms of major cells, array items are ordered in memory by major cells, by major cells within major cells, etc. Due to CPU caching on modern architectures, sequential memory access is more efficient than non-sequential access, and for good performance data design needs to take the order into account (favoring inverted tables [Hui 2018b], for example). In contrast, the Iliffe vector representation [Iliffe 1961] of a rank-`n` array (for `n≥2`) specifies a vector of pointers to rank-`(n-1)` subarrays (hello, major cells); as a result, data elements in different rows (cells) can be arbitrarily far from each other in memory. Iliffe vectors are implemented in Java, Perl, Python, and other programming languages.

Another representation is "strided representation" [Abrams 1970, §III; Munsey 1977; Nickolov 2013], where the ravelled data items are accessed through the function

```
array[⊂⍵] ↔ (,array)[offset+⍵+.×stride]
```

Here, `⍵` are the usual indices `i,j,k,` ... in each dimension, `offset` is an integer scalar, and `stride` is an integer vector with length the rank of `array`. This is a generalization of the usual index function where `offset←0` and `stride←×⍀⍨⌽1↓1,⍤¯1⍴array` or, equivalently,

```
array[⊂⍵] ↔ (,array)[(⍴array)⊥⍵].
```

Strided representation permits terse descriptions of the selection functions *take*, *drop*, *reverse*, *rotate*, and *transpose*, which can be effected by computing the shape of the result and appropriate values for `offset` and `stride`. For example, the dreaded dyadic *transpose* `⍺⍉⍵` obtains as follows [Abrams 1970, p.73; Berry 1979, p.146; Hui 1987, §3.1; Hui 2016a, §30]:

```
transpose←{
  shape  ← ⍺[⍋⍺] {⌊⌿⍵}⌸ (⍴⍵)[⍋⍺]
  stride ← ⍺[⍋⍺] {+⌿⍵}⌸ (×⍀⍨⌽1↓1,⍤¯1⍴⍵)[⍋⍺]
  offset ← 0
  (,⍵)[offset+(↑,⍤¯1⍳shape)+.×stride]
}

   x←    2 1 2 0 1
   y← ? 11 7 2 5 3 ⍴ 100
   (x⍉y) ≡ x transpose y
1
```

The last line of `transpose` produces the row-major representation from the strided representation. The computation of the strided representation itself is $O(\neq\alpha)$, or $O(\neq\rho\omega)$ since `(≠⍺)=≠⍴⍵`.

Strided representation is useful for application of compiler techniques. For *take*, *drop*, etc., you'd manipulate the representation until you get to a point where you actually need all the array items, and at that point you convert the strided representation into the row-major representation.

9.2 Small-Range Data

An array is *small-range* if the difference between its maximum and minimum values is small relative to its size. Faster algorithms are possible for small-range data in the important *index-of* (§2.2), *grade*, and *sort* (§2.5) functions. Benchmarks on small-range *v* big-range data can be used to evaluate the efficiency of APL implementations. For example, if these functions run at the same speed for small-range and big-range arguments, then some performance improvements remain to be exploited.

An array with a data type of a small domain (e.g. 1- or 2-byte integers or characters) can be trivially determined to be small-range. On larger domains, the range can be computed efficiently on modern computer architectures "on the fly", and small-rangeness can be exploited by interpreters without the complication and expense of array predicates. For example:

```
   x2←?9e5⍴3e4 ⋄ y2←?1e6⍴3e4 ⋄ (⌈⌿-⌊⌿)x2,y2  ⍝ 2-byte integers
29999
   xs←2e9+x2   ⋄ ys←2e9+y2   ⋄ (⌈⌿-⌊⌿)xs,ys  ⍝ 4-byte ints, small range
29999
   xb←1e4×x2   ⋄ yb←1e4×y2   ⋄ (⌈⌿-⌊⌿)xb,yb  ⍝ 4-byte ints, big   range
299990000
```

(The x variables are `9e5`-element vectors; the y variables are million-element vectors.)

```
   cmpx 'x2⍳y2' 'xs⍳ys' 'xb⍳yb'                ⍝ index-of (§2.2)
  x2⍳y2 → 2.77E¯3 |     0% □□
  xs⍳ys → 3.50E¯3 |   +26% □□
  xb⍳yb → 4.65E¯2 | +1583% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
   cmpx '⍋y2' '⍋ys' '⍋yb'                      ⍝ grade (§2.5)
  ⍋y2 → 1.38E¯2 |    0% □□□□□□□□□□□
  ⍋ys → 1.40E¯2 |   +1% □□□□□□□□□□□
  ⍋yb → 3.78E¯2 | +174% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
   cmpx '{⍵[⍋⍵]}y2' '{⍵[⍋⍵]}ys' '{⍵[⍋⍵]}yb'   ⍝ sort (§2.5)
  {⍵[⍋⍵]}y2 → 2.70E¯3 |    0% □□□□□
* {⍵[⍋⍵]}ys → 2.91E¯3 |   +7% □□□□□
* {⍵[⍋⍵]}yb → 1.77E¯2 | +554% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

`cmpx` is a utility function which compares execution times [Scholes 2019c]. It executes the expressions in its argument and, for each expression, displays the number of seconds required, the difference in time relative to the first expression, and a bar chart of the number of seconds. An `*` in the left margin indicates that the result of the expression differs from that for the first expression.

The last two benchmarks illustrate that "sorting is faster than grading" [Hui 2006].

9.3 Idioms (Special Code)

An idiom in a natural language, such as "raining cats and dogs" or "staircase wit" (itself from the French idiom "l'esprit de l'escalier"), is a phrase whose meaning is not predictable from the usual meanings of its constituent words. In constrast, as used in APL, an idiom is merely a common phrase, whose meaning is precisely as indicated by its parts [Perlis and Rugaber 1977; FinnAPL 2015]. Because APL idioms are commonly used, APL interpreters tend to recognize them and implement them by special code for improved performance [Brown 1988; Dyalog 2014b; Hui and Iverson 2004, Ax.B]. They are then idiomatic in the sense that the performance is better than expected, in time, space, or precision. Examples:

Time. The train `(1 ⍳⍨ cmp)` finds the index of where the comparison `cmp` between the two arguments is first true. By definition `⍺(1⍳⍨cmp)⍵ ↔ 1⍳⍨(⍺ cmp ⍵)`. But the former is recognized and implemented by special code, which quits as soon as it finds a true comparison, whereas the latter uses general code, which compares the arguments *in toto* and then finds the first 1. The special code is competitive even if the comparison is true only near the end or is never true.

```
      x←?1e6⍴1e9

      cmpx '5e8 (1⍳⍨<) x' '1 ⍳⍨ 5e8<x'
 5e8 (1⍳⍨<) x → 5.49E¯7 |      0%
 1 ⍳⍨ 5e8<x   → 2.41E¯4 | +43777% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

      cmpx '2e9 (1⍳⍨<) x' '1 ⍳⍨ 2e9<x'
 2e9 (1⍳⍨<) x → 2.94E¯4 |   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
 1 ⍳⍨ 2e9<x   → 2.48E¯4 | -16% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Space: The phrase `{≢⍵}⌸x` (§3.3) counts the number of occurrences of the unique items in `x`. The operand `{≢⍵}` is recognized by the *key* operator `⌸` and is implemented by special code which is more parsimonious in space (and also in time) than the general code. (`wsreq` is a utility function which reports on the workspace required to execute an expression [Scholes 2019f].)

```
      x←(⍳100),?1e6⍴100
      wsreq '{≢⍵}⌸x'
2052
      wsreq '{≢⊢⍵}⌸x'
9130908

      cmpx '{≢⍵}⌸x' '{≢⊢⍵}⌸x'
 {≢⍵}⌸x  → 7.60E¯4 |    0% ⎕⎕⎕⎕
 {≢⊢⍵}⌸x → 6.15E¯3 | +708% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Precision. The phrase `*○` ($e^{\pi z}$) is recognized by the parser and implemented by special code. The idiom recognition can be defeated by circumlocution, whence the phrase is computed by the usual code, subject to the vicissitudes of finite-precision floating-point computation.

```
      *○0j1                       *(○0j1)
¯1                          ¯1J1.22465E¯16

      1+*○0j1                     1+*(○0j1)
0                           0J1.22465E¯16

      ⊢ a←3 4⍴⍳8
0 1 2 3
4 5 6 7

      *○ 0j1×a÷2                  * (○0j1×a÷2)
1 0J1 ¯1 0J¯1               1              6.12323E¯17J1  ¯1J1.22465E¯16 ...
1 0J1 ¯1 0J¯1               1J¯2.44929E¯16 3.06162E¯16J1  ¯1J3.67394E¯16 ...
```

9.4 Magic Functions

A *magic function*, also known as a PDF, primitive defined function [Falkoff 1991, p.424] or predefined function [Brown 2017], is an APL-coded computation in the interpreter source code. Magic functions have long been in the APL folklore and used as an implementation technique, for example in micro-coded APL for the IBM 370/145 *circa* 1972 [Moore 2017] and in NARS *circa* 1981 [Smith 2018]. More recently, magic functions coded as dfns (§1.2) were used in over 100 places in the Dyalog APL interpreter to implement primitive functions and operators [Hui 2015a].

For example, as mentioned in §2.2, the expression `α∧.=⍤1 ω` where `α` and `ω` are matrices with the same number of columns, has been the target of optimization efforts since the 1970s [Bernecky 1977]. When `⍳` was extended, it was realized that `α∧.=⍤1 ω` is equivalent to `(≠α)(↑∘.=↓)α⍳α⍪ω`, converting a cross-tabulation on vectors to an outer product on integers. With a magic function an improvement by a factor of 5 or more was implemented in short order [Hui 2014, §12; Hui 2015a].

Magic functions as a tool of implementation exhibit Iverson's five important characteristics of notation in *Notation as a Tool of Thought* [Iverson 1980]:

- ease of expression
- suggestivity
- subordination of detail
- economy
- amenability to formal manipulations

9.5 assert

`assert` is an APL utility function used extensively in the Dyalog APL QA suite, executed daily to check for errors in the APL interpreter.

```
assert ← {⍺←'assertion failure' ⋄ 0∊⍵:⍺ ⎕signal 8 ⋄ shy←0}
```

The definition is subtle and full of technical arcana; the main thing is that propositions are stated in the positive sense. When an assertion failure occurs in a user definition, execution is suspended within that definition to allow inspection of the local state.

```
      assert 4 = 2+2
      assert 0 = 1+*○0j1
      assert 'deer' ≡ 'horse'
assertion failure
      assert'deer'≡'horse'
      ^
```

The `assert` utility has counterparts as C macros in APL interpreters, used 700 times in Dyalog APL and 1000 times in J. Again, the main thing is that the required condition is stated in the positive sense:

```
#define ASSERT(p,stmt)  {if(!(p)){stmt;}}
ASSERT(2>=r, ERANK);
ASSERT(xn==yn, ELENGTH);
ASSERT(t==INT1||t==INT2||t==INT4, EDOMAIN);
```

Absent `ASSERT`, these would be coded variously, repeated hundreds of times:

```
if(!(2>=r))ERANK;
if(xn!=yn)ELENGTH;
if(!(t==INT1||t==INT2||t==INT4))EDOMAIN;

if(2<r)ERANK;
if(!(xn==yn))ELENGTH;
if(t!=INT1&&t!=INT2&&t!=INT4)EDOMAIN;
```

`assert` was specified as an APL control structure in the 1970s [Johnston 1977] and more recently implemented as such in J [Hui 2000].

10 CURRENTS AND EDDIES

10.1 Terminology

Starting no later than 1972, Iverson used terms from natural languages instead of from mathematics or computer science [Hui 2010b]. From *Algebra: An Algorithmic Treatment* [Iverson 1972a, p.1]:

> Because algebra is a language, it has many analogies with English. These analogies can be helpful in learning algebra, and they will be noted and explained as they occur. For instance, the integers or counting numbers (1, 2, 3, 4, 5, 6, ...) in algebra correspond to the concrete nouns in English, since they are the basic things we discuss, and perform operations upon. Furthermore, functions in algebra (such as `+` (plus), `-` (subtract), and `×` (times)) correspond to the verbs in English, since they *do* something to the nouns. Thus, `2+3` means "add 2 to 3"; and `(2+3)×4` means "add 2 to 3 and then multiply by 4". In fact, the word "function"; (as defined, for example, in the *American Heritage Dictionary*), is descended from an older word meaning, "to execute", or "to perform".

Natural language terms were then used consistently in *A Dictionary of APL* [Iverson 1987] and the practice was carried over in the J dialect [Hui and Iverson 2004]:

noun	array
verb	function
adverb	operator
alphabet	character set
word formation	lexing
punctuation	control structures, parentheses, etc.
sentence	expression
dictionary	reference manual
epigram	one-liner ...

The natural language terms are an extended and useful metaphor, and are more readily understandable to a wider audience. For example, there is usually a hiccup when explaining *operator* to a non-programmer, a non-APL programmer, or a mathematician: in APL, an operator is different from a function; it's an operator in the sense of Heaviside; it's a higher-order function; etc. An *adverb*, however, can be more readily grokked: think quickly, think slowly, think sensibly, think originally (the action of a verb is modified by an adverb); read quickly, write quickly, think quickly (an adverb induces a family of actions).

Iverson named APL and J primitives carefully, choosing names which are apt, short, and mnemonic, for example, *floor*, *ceiling*, *ravel*, *nub*. In particular, *floor* and *ceiling* have come into general use, for example in *The Art of Computer Programming* [Knuth 1968, §1.2.4] and *Concrete Mathematics* [Graham et al. 1989, Ch. 3]. An exchange from the early days [Iverson 1964] is illuminating:

> Brooker: It is not obvious to me that these two symbols for FLOOR and CEILING have a great deal of mnemonic value.
> Iverson: Yes, but once you have read it, you can remember it.

He was annoyed to hear talk of the adicity or arity of a function instead of its valence because "adicity" and "arity", like "supercalifragilisticexpialidocious", are all affixes and rootless.

Finally, in a feat of "archæological algorithmic analysis" [Astrachan 2003], it was discovered that Iverson likely invented the term "bubble sort", although he did not invent the algorithm. (He probably wouldn't want to admit it if he did ☺.) From *A Programming Language* [Iverson 1962, p.217]:

> The result is to bubble each item upward in the sequence until it encounters an item with a smaller (or equal) key and then to leave it and continue bubbling the new smaller item. In particular, the smallest item is bubbled to the top.

One can believe that Iverson invented the term. It is consistent with his use of previously non-technical but descriptive, picturesque, and mnemonic terms in technical subjects.

> If one formerly contemplated [imaginary numbers] from a false point of view and therefore found a mysterious darkness, this is in large part attributable to clumsy terminology. Had one not called $+1$, -1, $\sqrt{-1}$ positive, negative, or imaginary (or even impossible) units, but instead, say, direct, inverse, or lateral units, then there could scarcely have been talk of such darkness.
>
> — C.F. Gauss [Gauss 1831, p.638], translated in [Ewald 1996, p.313]

10.2 What's in a Name?

An Implementation of J, Appendix A. Incunabulum [Hui 1992a, p.74], says:

> One summer weekend in 1989, Arthur Whitney visited Ken Iverson at Kiln Farm and produced—on one page and in one afternoon—an interpreter fragment on the AT&T 3B1 computer. I studied this interpreter for about a week for its organization and programming style; and on Sunday, August 27, 1989, at about four o'clock in the afternoon, wrote the first line of code that became the implementation described in this document.
>
> Arthur's one-page interpreter fragment is as follows: ...

Shortly thereafter, it became necessary to save the source file (file, singular) for the first time, and the name "J" was chosen.

Years later, Iverson said that he would have named it "APL0" but for the fact that the name was already taken by Alan Graham [Graham 1989; Iverson 2004]. (Graham later renamed his language "0"; neither APL0 nor 0 progressed very far.) At the time, APL2 was all the rage, and there was talk of the predecessors being APL1 [Brown 1984]. Possibly APL0 was supposed to be evocative of "APL back to its origins". There were no other attempts to pick another name because seemingly there were better things to do.

Not having "APL" as part of the name turned out to be consequential. It was one of the arguments, spoken and unspoken, supporting the proposition that "J is not APL" [Barman and Camacho 1991; Christensen 2006]. You can have a serious and reasoned discussion on whether J is APL, as in, if J is not APL, then what is APL? What makes APL APL [Hui and Kromberg 2020]? But such discussions usually emit radiation with wavelengths longer than 7000 Å rather than shorter.

Another consequence is that "J" is problematic in search engine queries. In August 1989 that was not a problem; if you were naming a language today you would have to give this serious consideration.

Speaking of names, another language claimant to "APL" emerged in September 2018, Alexa Presentation Language [Amazon 2018], reminiscent of the struggle in the 1960s when APL won the battle for its name [McDonnell 1981b].

> Before APL was called APL, it was called "Iverson notation". Ken mused that it should be called simply "the notation". After all, we don't say "God's grass", just "the grass".
>
> — Paul Berry, 1987 [Hui 2020a]

10.3 Backward Compatibility

APL is a programming language with a 50-year history in continuous commercial use. It would be surprising (and impressive!) if current APL implementations were completely backward compatible. Some dialects (e.g. APL2, Dyalog APL) strive to be as backwardly compatible as possible. Other dialects (e.g. J, k), boldly going where no APL dialect has gone before, are not compatible but preserve the essential ideas of APL.

Sometimes, even dialects that strive for compatibility make a "breaking change" when the reasons for it are judged to be strong, stronger than the reasons for the status quo. An example is the change in the definition of `⍺*⍵` to `*⍵×⊗⍺` when complex numbers were introduced [Penfield 1979; Penfield 1981; McDonnell 1981a]. Previously, the expression `¯8*÷3` had value `¯2` but now is `1j1.73205`, the value of `*(÷3)×⊗¯8` and the principal cube root of `¯8`. It turned out that before complex numbers were introduced, the `¯2` value obtained by a procedure used when `⍺` was negative: `⍵` was expressed as a rational number `m÷n` in lowest terms, and if `n` was odd then `⍺*⍵` gave a real result (instead of signalling error). The negative real value result, subject to the vicissitudes of finite-precision floating-point representation and calculation, was more a *tour de force* and a sort of Easter egg left by the *APL\360* language implementer than mathematically apt.

A possible non-compatible change is the extension of `∈` to higher-ranked arguments (§2.2), which requires that it *not* treat the right argument as if it were ravelled. Since `∈` is widely used, it remains to be seen whether this change will be made.

For *power* and *member*, an error was replaced by a result and such replacement posed difficulties for future extensions. Premature optimization is said to be the root of all evil in programming [Knuth 1974, p.671], but premature generalization is worse. Premature optimizations can at least be backed off.

10.4 Index Origin

Few topics in APL are as apparently trivial but at the same time as contentious (including between the authors) and practically fraught with implications, as the question of index origin [Hui 2010e]: Does indexing start from 1 or from 0? This paper uses 0-origin, with the exception of the examples immediately below on the right and the example from 1967 in §11.7.

```
   ⎕io←0
   ⍳5
0 1 2 3 4
   'paronomasiac'⍳'aeiou'
1 12 9 3 12
   'loves' 'me' 'not' [1 2]
┌──┬───┐
│me│not│
└──┴───┘
```

```
   ⎕io←1
   ⍳5
1 2 3 4 5
   'paronomasiac'⍳'aeiou'
2 13 10 4 13
   'loves' 'me' 'not' [1 2]
┌─────┬──┐
│loves│me│
└─────┴──┘
```

The problem is `⎕io`, a system variable which is an implicit argument to a few primitives—few, but important and ubiquitous. `⎕io` can be dynamically localized and can be set to 0 or 1. Code which needs to work in either index origin would have `⎕io` sprinkled hither, thither, and yon. It is, in the words of Iverson, a feature more defended against than used [Iverson 2004].

Multiple and ingenious arguments can be and have been trotted out to support one or the other value. (Including in the non-APL world, for example [Dijkstra 1982; Cassani and Conway 2018].) One of us (Hui), since he is "holding the pencil", or one of the two pencils, feels entitled to present one argument for 0-origin: In Iverson's Turing Lecture [Iverson 1980] which used 1-origin, many APL expressions would be simplified if 0-origin were used instead.

It would have been preferable to fix the index origin at either 1 or 0, and get rid of `⎕io` altogether. Those in the "wrong" camp would just have to grit their teeth and get on with it. But backward compatibility precludes cutting this Gordian knot.

`⎕io` *delenda est*!

10.5 APL Characters

A history is more than a dispassionate recounting of major events, more than a recording of the progression from highlight to highlight (with a few lowlights in between). People are involved. People passionate, witty, argumentative, messy, glorious. Here, we provide a glimpse of the APL community, what they care about, what they fight over, what they are like as people, etc.

It has been said, e.g. by Fred Brooks [Falkoff and Iverson 1978], that APL has quite a character set. *APL Quotations and Anecdotes* [Hui 2020a] is a collection of over 180 stories by or about these characters. Seven stories from the collection:

- Ken regularly stood up to Howard Aiken, and Aiken stood six-foot-three and had Spockean ears and pointed tufts of hair and when he reared up and looked down at you, you thought you were looking at the Devil. And when he growled at you, you thought that too. Never mind: Aiken had the characteristic that if you stood up to him he respected that, and if you didn't he just trod you underfoot. Ken stood up to him.
  Ken's manner was always argumentative, but argumentative with an effective expression: "Perhaps one ought to think about it this way ..." This was in contrast with Aiken's "Goddamit it's gotta be so-and-so, all right?" Ken was firm in his views but he was reasonable. He satisfied the old Latin motto, *Numquam incertus, semper apertus*—never uncertain, always open.

  — Fred Brooks, *The Language, The Mind and the Man* [Brooks Jr. 2006]

- Years ago, I (was) volunteered to give an impromptu "what's APL" elevator pitch to some Microsoft engineers. I stood up and started:
  "Um, APL's only data type is the array ..."
  "Er, APL's primitive functions take arrays as arguments ..."
  Someone in the audience piped up: "That's just as well, then."

  — John Scholes

- Years later, talking about "the one-page thing" [§10.2], Arthur Whitney quipped that he still wanted to do it in one page, but using *n*-point font. (The value of *n* increases with each retelling of the story.)

  — Roger Hui

- Reasons for Liking `⍟`
  - It's kind of cute, possessing a radial symmetry.
  - It denotes a function for which conventional mathematical notation [Abramowitz and Stegun 1964, §4.1] does not have a good symbol:
    `⍟y` ↔ ln y or log y
    `x⍟y` ↔ $\log_x y$
  - It alludes to `0=1+⋆○0j1`, the most beautiful equation in all of mathematics [Hui 2010c], relating in one short phrase the fundamental quantities 0, 1, e, π, and `0j1` and the basic operations plus, times, and exponentiation.
  - It is a visual pun—the symbol looks like the cross section of a felled tree, i.e. a log [McDonnell 1977].

  — Roger Hui, *My Favorite APL Symbol* [Hui 2013a]

- Ken visited us now and again and I remember once I was struggling with translating his *An Introduction to APL* into Danish. I told Ken that I was completely unable to come up with a good translation for the APL function *ravel*, which had previously suffered the prosaic name "make list of" in Danish. [He said,] "You should use the word you use when you have knitted something and you then undo the knitting—when you are removing the structure and are left with the thread." Suddenly the term *ravel* made sense to me in a completely different way.

  — Gitte Christensen [Christensen 2006]

- In school and in textbooks it is proven that sorting requires $O(n \times \circledast n)$ comparisons. I suppose that is why Bob Smith was initially skeptical when I told him that in J and Dyalog APL you can sort a real vector in $O(n)$ time.
  I opined to Dyalog colleagues that Bob is a smart guy and if you tell a smart guy that something is possible (and he believed you) then he would soon figure out how to do it. He did.

  — Roger Hui

- In January 1999, Ken and Jean Iverson visited Eugene McDonnell in Palo Alto. Eugene held a dinner party in their honor on the thirteenth. In attendance were Eugene and Jeanne McDonnell, Ken and Jean Iverson, Arthur Whitney and Janet Lustgarten, Jim and Karen Brown, Paul and Sachiko Berry, Charles Brenner and Sarita Berry, Larry and Beverly Breed, Harry Saal, Ken's nephew Derrick Iverson and his wife and new baby, David Steinbrook, Joel Kaplan, Dick Dunbar, Joey Tuttle (who flew in from Boston), and one more.
  One of the events of the party was to have been a telephone call from Kyosuke Saigusa in Japan. During the party, Eugene went to make the phone call to Japan as prearranged. He came back a few minutes later, perplexed, saying that he only managed to reach Mrs. Saigusa, who said that Mr. Saigusa was not available.
  At that point Saigusa-san walked in through the front door, explaining that he was not available by phone from Japan because he was there in Palo Alto.

  —Joey Tuttle

11 CODE

> The mathematician's patterns, like the painter's or the poet's must be *beautiful*; the ideas like the colours or the words, must fit together in a harmonious way. Beauty is the first test: there is no permanent place in the world for ugly mathematics.
>
> — G.H. Hardy, *A Mathematician's Apology* [Hardy 1940, p.14]

> The other day I was reading a newspaper, an article by somebody in the arts who said if Shakespeare were alive today he'd be writing for TV. And I said to myself when I read that, "Not so. If Shakespeare were alive today, he'd be a programmer, and he'd be writing one-liners in APL."
>
> — Alan Perlis [Perlis 1978]

We conclude the paper by looking at some APL code. These are solutions to benchmark problems dating from up to 50 years ago, used to assess the goodness of language use and language design. Readers not conversant in APL may find them difficult to understand in detail; nevertheless, it is still fruitful to compare and contrast solutions in other programming languages.

Are these "monument-quality" code, fit for presentation to the Galactic Emperor [Hui 2010a]? Some are, some not yet. For the latter to achieve that quality requires improvements to APL implementation or the APL language itself.

11.1 Random Numbers

Generate `⍵` random numbers selected from `⍳≠⍺` according to the weights `⍺`, a vector of positive real numbers [Hui 2017d]. For example, if `a←7 5 6 4 7 2 0.4` are the weights, then in `a ran 1e6` the number `0` should occur roughly as often as `4` (both with a weight of `7`) and 3.5 times as often as `5` (with a weight of `2`).

```
      ran ← {(0;+\⍺÷+/⍺)⍸?⍵⍴0}

      a← 7 5 6 4 7 2 0.4
      ≢ t←a ran 1e6
1000000
      10↑t
3 3 0 3 4 6 1 2 4 4
      +/ t ∘.= ⍳≠a     ⍝ observed # of occurrences
223502 159090 191022 127226 223008 63613 12539
      1e6 × a÷+/a      ⍝ expected # of occurrences
222930 159236 191083 127389 222930 63694.3 12738.9
```

The technique can be used to generate random numbers according to a probability distribution [Abramowitz and Stegun 1964, §26.8]. If a discrete distribution with values `v` and probabilities `p`, then `v[p ran ⍵]`. If a continuous distribution, convert it into a discrete one by dividing `(¯∞,∞)` into an appropriate number of intervals. The interval midpoints are the values; the probabilities are the differences of the cumulative distribution function at the interval end points.

The problem was solved in 1975 [IPSA 1975]; the current solution uses to advantage the extension of `?0` to generate a random number drawn uniformly from the open interval $(0, 1)$ (suggested by Whitney) and the *interval index* function `⍸` (§2.3).

11.2 Quicksort

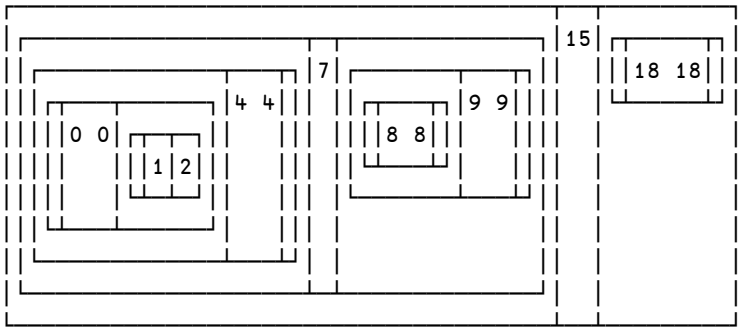
Quicksort works by choosing a "pivot" at random among the major cells, then catenating the sorted major cells which strictly precede the pivot, the major cells equal to the pivot, and the sorted major cells which strictly follow the pivot, as determined by a comparison function `⍺⍺`. Defined as an operator `Q`:

```
      Q←{1≥≠⍵:⍵ ⋄ s←⍵ ⍺⍺ ⍵[]⍨?≠⍵ ⋄ (∇ ⍵≠⍨0>s),(⍵≠⍨0=s),(∇ ⍵≠⍨0<s)}

      ⍝ precedes            ⍝ follows             ⍝ equals
      2 (×-) 8              8 (×-) 2              8 (×-) 8
¯1                     1                      0

      numv← 2 0 4 7 15 9 8 0 4 9 18 8 1 18
      (×-)Q numv
0 0 1 2 4 4 7 8 8 9 9 15 18 18
      cmp←{-⍒(⍋⍺⍵)×⍺≠⍵}
      'alpha' cmp 'beta'    ⍝ precedes
¯1
      'beta'  cmp 'alpha'   ⍝ follows
1
      'beta'  cmp 'beta'    ⍝ equals
0

      t←' the heart has its reasons that the reason does not know'
      ⊢ words← ' ' (≠⊆⊢) t
┌───┬─────┬───┬───┬───────┬────┬───┬──────┬────┬───┬────┐
│the│heart│has│its│reasons│that│the│reason│does│not│know│
└───┴─────┴───┴───┴───────┴────┴───┴──────┴────┴───┴────┘
      cmp⍨ Q words
┌────┬───┬─────┬───┬────┬───┬──────┬───────┬────┬───┬───┐
│does│has│heart│its│know│not│reason│reasons│that│the│the│
└────┴───┴─────┴───┴────┴───┴──────┴───────┴────┴───┴───┘

      cmp⍨ Q numv
0 0 1 2 4 4 7 8 8 9 9 15 18 18
      m                        cmp⍨⍤1 Q m
3 1 4 1 5 9              1 4 2 8 5 7
2 7 1 8 2 8              2 7 1 8 2 8
1 4 2 8 5 7              2 7 1 8 2 8
3 1 4 1 5 9              3 1 4 1 5 9
2 7 1 8 2 8              3 1 4 1 5 9
```

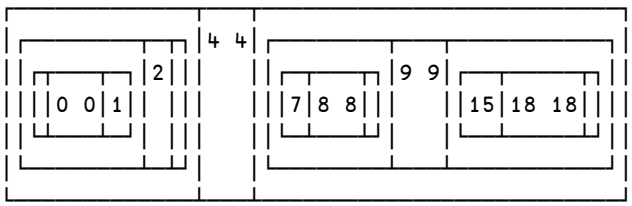
The above formulation is not new; see for example Figure 3.7 of the classic *The Design and Analysis of Computer Algorithms* [Aho et al. 1974]. However, unlike the pidgin ALGOL program in Figure 3.7, `Q` is executable, and the partial order used in the sorting is an operand, the `(×-)` and `cmp⍨` and `cmp⍨⍤1` in the examples above.

`Q3` is a variant that catenates the three parts enclosed instead of the parts *per se*.

```
Q3←{1≥≠⍵:⍵ ⋄ s←⍵ ⍺⍺ ⍵[]⍨?≠⍵ ⋄ (⊂∇ ⍵≠⍨0>s),(⊂⍵≠⍨0=s),⊂∇ ⍵≠⍨0<s}
(×-)Q3 numv
```




```
(×-)Q3 numv
```



Applying the function derived from `Q3` on the same argument multiple times gives different results because the pivots are chosen at random. In-order traversal of the results does yield the same sorted array.

11.3 Permutations

Generate the sorted matrix of all permutations of `⍳⍵` [Hui 2016a, §19; McDonnell 2003; Hui 2015b].

```
   perm 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

```
   ⍴ perm 3
6 3
```

`perm ⍵` obtains by indexing each row of the matrix `⍸⍤1 ∘.=⍨ ⍳⍵` by `0,1+perm ⍵-1`. For example, for `⍵=4`:

```
   ∘.=⍨ ⍳4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

```
   ⍸⍤1 ∘.=⍨ ⍳4
0 1 2 3
1 0 2 3
2 0 1 3
3 0 1 2
```

```
   ⊢ p1←0,1+perm 4-1
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
```

```
      ⊂⍤¯1 ⊢ (⊂⍳4) ⌷⍤1 ⍋⍤1 ∘.=⍨ ⍳4  ⍝ apply ⊂⍤¯1 for a compact display
┌─────────┬─────────┬─────────┬─────────┐
│0 1 2 3│1 0 2 3│2 1 0 3│3 0 1 2│
│0 1 3 2│1 0 3 2│2 1 3 0│3 0 2 1│
│0 2 1 3│1 2 0 3│2 0 1 3│3 1 0 2│
│0 2 3 1│1 2 3 0│2 0 3 1│3 1 2 0│
│0 3 1 2│1 3 0 2│2 3 1 0│3 2 0 1│
│0 3 2 1│1 3 2 0│2 3 0 1│3 2 1 0│
└─────────┴─────────┴─────────┴─────────┘
```

Putting it all together:

```
perm ← {0=⍵:1 0⍴0 ⋄ ((!⍵),⍵)⍴(⊂0,1+∇ ¯1+⍵)⌷⍤1 ⍋⍤1 ∘.=⍨ ⍳⍵}
```

```
   perm 4              ⍴ perm 4
0 1 2 3             24 4
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
1 0 2 3
1 0 3 2
   ...
3 1 0 2
3 1 2 0
3 2 0 1
3 2 1 0
```

It is instructive to compare the modern dfns rendition of the algorithm against a tradfn from 1981 [Hui 1981].

```
 p←perm n;i;ind;t
⍝ all permutations of ⍳n
 p←(×n,n)⍴⎕io
 →(1≥n)⍴0
 t←perm n-1
 p←(0,n)⍴ind←⍳n
 i←n-~⎕io
L10:p←(i,((i≠ind)/ind)[t]),[⎕io]p
 →(⎕io≤i←i-1)⍴L10
```

The comparison illustrates use of dfns *v* tradfns (§1.2), the expressive power of the *rank* operator ⍤ (§3.1), the benefits of denoting indexing by a single symbol ⌷ (§2.4), and the mental and aesthetic cost of ⎕io (§10.4). It also illustrates a drawback of **Enclose-Reduction-Style** *v* **Insert-Reduction-Style** [§4.2; ISO/IEC 2001, §9.2.1]: with the latter, the phrase ⍪≠ can replace ((!⍵),⍵)⍴ in the dfn version of perm (and similarly replace (2 1*⍨⍴⍵)⍴ in SG in §3.2).

11.4 Symmetries of the Square

A permutation can be represented as an integer vector or as a square Boolean matrix with exactly one 1 in each row and each column, that is, a permutation matrix. Functions `pm←⊢∘.=⍳∘≢` and `mp←⍳∘1⌸1` transform from one to the other [Iverson 1980, §3.3].

```
   p                      a
6 3 2 1 5 4 0          96 84 59 5 19 47 36

   pm p
0 0 0 0 0 0 1             a[p]
0 0 0 1 0 0 0          36 5 59 84 47 19 96
0 0 1 0 0 0 0
0 1 0 0 0 0 0             (pm p)+.×a  ⍝ inner product
0 0 0 0 0 1 0          36 5 59 84 47 19 96
0 0 0 0 1 0 0
1 0 0 0 0 0 0

   mp pm p
6 3 2 1 5 4 0
```

`⊢ ⍋ ⍒ ⌽` on permutations produce permutation results. They can be identified with `⊢ ⍉ ⌽⍉ ⌽` on square matrices. Since `⊢ ⍉ ⌽⍉ ⌽` on matrices are transpositions of the square, then so are `⊢ ⍋ ⍒ ⌽` on permutation vectors.

```
   (⊢ pm p) ≡ pm ⊢p
1
   (⍉ pm p) ≡ pm ⍋p
1
   (⌽⍉pm p) ≡ pm ⍒p
1
   (⌽ pm p) ≡ pm ⌽p
1
```

`⊢` *identity function*
`⍋` *grade up* (ascending sort indices)
`⍒` *grade down* (descending sort indices)
`⌽` *reverse*
`⍉` *transpose*

```
   D8
```

⊢	⍒	⍒⍒	⍋⌽	⌽	⍋	⍋⍒	⍒⌽
⍒	⍒⍒	⍋⌽	⊢	⍒⌽	⌽	⍋	⍋⍒
⍒⍒	⍋⌽	⊢	⍒	⍋⍒	⍒⌽	⌽	⍋
⍋⌽	⊢	⍒	⍒⍒	⍋	⍋⍒	⍒⌽	⌽
⌽	⍋	⍋⍒	⍒⌽	⊢	⍒	⍒⍒	⍋⌽
⍋	⍋⍒	⍒⌽	⌽	⍋⌽	⊢	⍒	⍒⍒
⍋⍒	⍒⌽	⌽	⍋	⍒⍒	⍋⌽	⊢	⍒
⍒⌽	⌽	⍋	⍋⍒	⍒	⍒⍒	⍋⌽	⊢

`D8` is an 8-by-8 matrix of enclosed strings (character vectors) [Hui 1981; Hui 1987, §4.4; Hui 2016a, §13]. It is also a group table and a compact presentation of numerous identities involving `⊢ ⍋ ⍒ ⊖` on permutation vectors—`D8[i;0]` composed with `D8[0;j]` is `D8[i;j]`. For example:

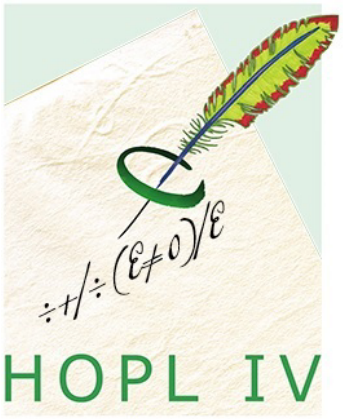
i	j	D8[i;0]	D8[0;j]	D8[i;j]
5	5	⍋	⍋	⊢
2	2	⍒⍒	⍒⍒	⊢
1	2	⍒	⍒⍒	⍋⊖
1	5	⍒	⍋	⊖
1	6	⍒	⍋⍒	⍋

That is, the `2 2` entry asserts that `⍒⍒⍒p ↔ ⊢p`; the `1 5` entry asserts that `⍒⍋p ↔ ⊖p`; and so on. The veracity of these assertions can be checked by

```
      (⍷¨D8,¨'p') ≡ ⍷¨(∘.,⍨⍳8)⌷D8),¨'p' ⊣ p←?⍨100
1
```

The expressions here would be less elegant and less effective as tools of thought if rendered as names (§6.5) or multicharacter symbols (§6.3) [Hui 2005d]. A version of `D8` using single symbols for the matrix functions `⊢ ⍉ ⌽ ⊖` appeared in 1964 [Iverson 1964, Table 9].

11.5 Total Resistance



A HOPL IV Badge
(Courtesy of Richard P. Gabriel and Guy L. Steele Jr.)

One of the HOPL IV badges has an APL expression on it: `÷+/÷(e≠0)/e`, the reciprocal of the sum of the reciprocals of the non-zero values of `e`. The expression computes the total resistance of components connected in parallel, whose resistance values are the vector `e`.

There is an alternative phrasing in modern APL: `+⌿⍤÷⍤~0`, sum *under* reciprocal (§3.5), without 0s. If arithmetic were extended to infinity (§4.6), in particular if `÷0 ↔ ∞` and `÷∞ ↔ 0`, then the expression would simplify to `+⌿⍤÷e`, without the *without* 0 (`~0`). It makes sense in terms of the electrical apparatus: when resistors are connected in parallel with one or more having 0 resistance, the total resistance should be 0.

A few other well-known computations can also be stated in the same pattern:

`+⌿⍤⍟`	sum under log	product
`+⌿⍤(*∘p)`	sum under power	L_p norm
`+⌿⍤(*∘2)`	sum under square	Euclidean norm

11.6 Ackermann Function

The Ackermann function is a fast growing function defined on non-negative integers. The presentation here is substantially as appeared in [Hui 1992b; Hui 2016a, §39].

```
ack←{
  0=⍺: 1+⍵
  0=⍵: (⍺-1) ∇ 1
  (⍺-1) ∇ ⍺ ∇ ⍵-1
}
   (⍳4) ∘.ack ⍳8
1  2  3  4   5   6   7    8
2  3  4  5   6   7   8    9
3  5  7  9  11  13  15   17
5 13 29 61 125 253 509 1021
   4 ack⍨ 0 1
13 65533
```

We present a lemma and its proof. The proof consists of a list of expressions, each equivalent to its precedessor and anotated by evidence for the equivalence [Iverson 1980].

Lemma: If `⍺ ack ⍵` ↔ `f⍤(3∘+)⍵` for a function `f`, then `(1+⍺)ack ⍵` ↔ `f⍤(1+⍵)⍤(3∘+)1`.

Proof: By induction on `⍵`.

`(1+⍺) ack 0`	basis
`⍺ ack 1`	definition of `ack`
`f⍤(3∘+)1`	antecedent of lemma
`f⍤(1+0)⍤(3∘+)1`	`⍤` (§3.2)
`(1+⍺) ack ⍵`	induction
`⍺ ack (1+⍺) ack ⍵-1`	definition of `ack`
`f⍤(3∘+) (1+⍺) ack ⍵-1`	antecedent of lemma
`f⍤(3∘+) f⍤(1+⍵-1)⍤(3∘+)1`	inductive hypothesis
`¯3∘+ f 3∘+ ¯3∘+ f⍤⍵ ⊢3∘+ 1`	`⍤` (§3.5) and `⊢`
`¯3∘+ f f⍤⍵ ⊢3∘+ 1`	`+`
`¯3∘+ f f⍤(1+⍵) ⊢3∘+ 1`	`⍤`
`f⍤(1+⍵)⍤(3∘+)1`	`⍤`
	QED

Using the lemma (or otherwise), it can be shown that:

`0∘ack = 1∘+⍤(3∘+)`	`⍝` *successor*
`1∘ack = 2∘+⍤(3∘+)`	`⍝` *plus*
`2∘ack = 2∘×⍤(3∘+)`	`⍝` *times*
`3∘ack = 2∘*⍤(3∘+)`	`⍝` *power*
`4∘ack = *⍟∘(⍴∘2)⍤(3∘+)`	`⍝` *power tower*
`5∘ack = {*⍟∘(⍴∘2)⍤(1+⍵)⍤(3∘+) 1}`	`⍝` *power tower, powered*

11.7 Ken Iverson's Favorite APL Expression?

Well, we don't know what Ken Iverson's favorite APL expression was or if he even had a favorite APL expression. But we can guess. From *A History of APL in 50 Functions* [Hui 2016a, §8]:

```
      ⊢ x←,1
1
      ⊢ x←(0,x)+(x,0)
1 1
      ⊢ x←(0,x)+(x,0)
1 2 1
      ⊢ x←(0,x)+(x,0)
1 3 3 1
```

The expression `(0,x)+(x,0)` or its commute, which generates the next set of binomial coefficients, is present in the document that introduced *APL*\360 [Falkoff and Iverson 1967, fig.1] and the one that introduced J [Hui et al. 1990, Gc&Gd]; in *Elementary Functions: An Algorithmic Treatment* [Iverson 1966, p.69&208], in *APL*\360 *User's Manual* [Falkoff and Iverson 1968, A.5], in *The Use of APL in Teaching* [Iverson 1969, p.19], in *Algebra: An Algorithmic Treatment* [Iverson 1972a, p.141], in *Introducing APL to Teachers* [Iverson 1972b, p.22], in *APLSV User's Manual* [Falkoff and Iverson 1973a, p.19], in *An Introduction to APL for Scientists and Engineers* [Iverson 1973, p.19], in *Elementary Analysis* [Iverson 1976, ex.1.68], in *Programming Style in APL* [Iverson 1978c, §6], in *Notation as a Tool of Thought* [Iverson 1980, A.3], in *Applied Mathematics for Programmers* [Iverson 1986a, p.38&102], in *Mathematics and Programming* [Iverson 1986b, p.61], in *A Dictionary of APL* [Iverson 1987, mVn], in *Programming in J* [Iverson 1991b, p.18], in *J Phrases* [Burke et al. 1996, p.14].

The following are two presentations of the ideas, [Falkoff and Iverson 1967, fig.1] on the left and [Hui 2016a, §8] on the right. The 1967 version required 1-origin indexing (§10.4).

```
    ∇ pascal n
[1]   j← 1 1
[2]   j
[3]   →0×⍳n=j[2]
[4]   j←(j,0)+0,j
[5]   →2
    ∇

   pascal 1
1 1
   pascal 2
1 1
1 2 1
   pascal 5
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
   j
1 5 10 10 5 1
```

```
   pascal← {(0∘,+,∘0)⍣⍵,1}

   pascal 0
1
   pascal 1
1 1
   pascal 2
1 2 1
   pascal 5
1 5 10 10 5 1

   pascal¨ ⍳6
┌─┬───┬─────┬───────┬─────────┬───────────┐
│1│1 1│1 2 1│1 3 3 1│1 4 6 4 1│1 5 10 10 5 1│
└─┴───┴─────┴───────┴─────────┴───────────┘

   pascal⍤0 ⍳6
1 0  0  0 0 0
1 1  0  0 0 0
1 2  1  0 0 0
1 3  3  1 0 0
1 4  6  4 1 0
1 5 10 10 5 1
```

A *APL*\360 ACKNOWLEDGMENTS

From *APL*\360 *User's Manual* [Falkoff and Iverson 1968]:

The APL language was first defined by K.E. Iverson in *A Programming Language* (Wiley, 1962) and has since been developed in collaboration with A.D. Falkoff. The *APL*\360 Terminal System was designed with the additional collaboration of L.M. Breed, who with R.D. Moore$^{\oplus}$, also designed the S/360 implementation. The system was programmed for S/360 by Breed, Moore, and R.H. Lathwell, with continuing assistance from L.J. Woodrum†, and contributions by C.H. Brenner, H.A. Driscoll$^{\pm}$, and S.E. Krueger$^{\pm}$. The present implementation also benefitted from experience with an earlier version, designed and programmed for the IBM 7090 by Breed and P.S. Abrams$^{\otimes}$.

The development of the system also profited from ideas contributed by many users and colleagues, notably E.E. McDonnell, who suggested the notation for the signum and the circular functions.

In the preparation of the present manual, the authors are indebted to L.M. Breed for many discussions and suggestions; to R.H. Lathwell, E.E. McDonnell, and J.G. Arnold†† for critical reading of successive drafts; and to Mrs. G.K. Sedlmayer and Miss Valerie Gilbert for superior clerical assistance.

A special acknowledgment is due to John L. Lawrence, who provided important support and encouragement during the early development of APL implementation, and who pioneered the application of APL in computer-related instruction.

$\oplus$ I.P. Sharp Associates, Toronto, Canada.
$\dagger$ General Systems Architecture, IBM Corporation, Poughkeepsie, N.Y.
$\pm$ Science Research Associates, Chicago, Illinois.
$\otimes$ Computer Science Department, Stanford University, Stanford, California.
$\dagger\dagger$ Industry Development, IBM Corporation, White Plains, NY.

(Foreground, L to R) Dick Lathwell, Ken Iverson, Roger Moore, Adin Falkoff, Phil Abrams, and Larry Breed, in the I.P. Sharp Associates hospitality suite during the 1978 APL Users Meeting in Toronto, Canada.
(Background, L) Jon McGrew.

B SUMMARY OF NOTATION

The following is a summary of the symbols used in the text. A complete reference can be found in http://docs.dyalog.com/17.0/Dyalog APL Language Reference Guide.pdf [Dyalog 2018a].

Functions

conjugate	+	*plus*	*ravel major cells*	⍪	*append*
negate	−	*minus*	*ravel*	,	*append last*
signum	×	*times*	*indices*	⍳	*index-of*
reciprocal	÷	*divide*	*indices from Boolean*	⍸	*interval index*
exponential	*	*power*	*enlist*	∈	*member*
natural log	⍟	*log*	-	⍷	*search*
magnitude	|	*residue*	*unique*	∪	*union*
floor	⌊	*min*	*materialize*	⌷	*index*
ceiling	⌈	*max*	-	[ ]	*index*
factorial	!	*binomial*	*disclose*	↑	*take*
π times	○	*circle*	*split*	↓	*drop*
-	=	*equal*	*grade up*	⍋	*grade up per* ⍺
-	≠	*not equal*	*grade down*	⍒	*grade down per* ⍺
(*box*)	<	*less than*	*transpose*	⍉	*transpose*
(*open*)	>	*greater than*	*reverse last*	⌽	*rotate last*
-	≤	*less or equal*	*reverse*	⊖	*rotate*
-	≥	*greater or equal*	*nest*	⊆	*partition*
-	∧	*and*	*enclose*	⊂	*partition*
-	∨	*or*	*disclose/first*	⊃	*pick*
-	⍲	*nand*	-	⊥	*base value*
-	⍱	*nor*	*identity*	⊣	*left*
not	~	*without*	*identity*	⊢	*right*
roll (*random*)	?	*deal*	*execute*	⍎	*execute*
shape	⍴	*reshape*	*format*	⍕	*format*
tally	≢	*not match*	*isolate*	⍪	-
depth	≡	*match*			

Monadic Operators

⌿	*reduce/replicate*
/	*reduce/replicate last*
⍀	*scan/expand*
\	*scan/expand last*
⍤	*each*
⍨	*commute*
⌸	*key*
⫴	*parallel*
⍰	*lazy* (*Schrödinger*)†
[ ]	*axis*

Dyadic Operators

⍤	*rank*
⍣	*power*
.	*inner/outer product*
∘	*compose/curry*
⍢	*under* (*dual*)†
⍛	*obverse*†
@	*at*
⌺	*stencil*

Other

⍝	comment
¯	minus sign (numbers)
←	assignment
⍬←	non-local assignment
⎕←	display in session log
→	goto
⋄	statement separator
⍺	left argument
⍵	right argument
⍺⍺	left operand
⍵⍵	right operand
∇	self-reference (function)
∇∇	self-reference (operator)
:	guard (dfn); label (tradfn)

Arrays

⍬	empty numeric vector
∞	infinity (number)†

Non-APL Notation

↔	equivalent
O( )	order of complexity
[,)	half open interval; etc.
$	*sparse*

† indicates not yet implemented

C A PARSER MODEL

The function `parse` models the interpretation of an APL expression. It applies to a character vector of an APL expression and produces a fully-parenthesized equivalent expression. For example:

```
      parse '+⌿⍤1 ⊢3 4⍴⍳12'
(((+⌿)⍤1)(⊢(3 4⍴(⍳12))))
```

The description and algorithm are adapted from *A Dictionary of APL* [Iverson 1987, §I and Table 2] and J [Hui and Iverson 2004, §II.E]. An alternative parser model can be found in [Scholes 2019e].

C.1 Description

The two main data structures in `parse` are a queue and a stack, each a 2-column matrix of expressions and class letters. The queue is initialized by the tokens of the argument, prefaced by a marker `⋄`. The stack is initialized as four markers. For example, for the argument expression `'*○0J1 2'`, the queue and the stack would be initialized as follows. (Typically, the transpose of the queue and the stack are presented for a more compact display.)

```
    ⍉Queue             ⍉Stack
 ┌─┬─┬─┬──────┐     ┌─┬─┬─┬─┐
 │⋄│*│○│0J1  2│     │⋄│⋄│⋄│⋄│
 ├─┼─┼─┼──────┤     ├─┼─┼─┼─┤
 │⋄│F│F│A     │     │⋄│⋄│⋄│⋄│
 └─┴─┴─┴──────┘     └─┴─┴─┴─┘
```

Eligibility for evaluation is determined only by the *class* of an element. In addition to the ordinary APL classes (array, functions, operators), ten classes internal to `parse` are used:

`A`	array	`N`	name
`F`	function	`I`	an "index expression", including `]`
`M`	monadic operator	`J`	an "index expression", including `[ ]`
`D`	dyadic operator	`⋄`	marker
			`←()[;]` *per se*

Parsing is controlled by the three tables `Cases`, `Actions`, and `Masks`.

```
    ⍕Cases                              Actions       Masks
 ⋄←([;       F     A     *       0     Fn1           0 1 1 0
 ⋄←([;]MFA   F     F     A       1     Fn1           0 0 1 1
 ⋄←([;]MFA   A     F     A       2     Fn2           0 1 1 1
 ⋄←([;]MFA   FA    M     *       3     Op1           0 1 1 0
 ⋄←([;]MFA   FA    D     FA      4     Op2           0 1 1 1
 ⋄←(]MFA     FA    F     F       5     T3            0 1 1 1
 ⋄←(         AFD   AFD   *       6     T2            0 1 1 0
 ⋄←([;]MF    AN    ←     DMFA    7     Assign        0 1 1 1
 (           DMFA  )     *       8     Paren         1 1 1 0
 ]           *     *     *       9     I0            1 0 0 0
 ;           I     *     *      10     I1            1 1 0 0
 ;           A     I     *      11     I2            1 1 1 0
 [           I     *     *      12     I3            1 1 0 0
 [           A     I     *      13     I4            1 1 1 0
 ⋄←([;]DMF   A     J     *      14     I5            0 1 1 0
 ⋄←([;]MFA   F     J     *      15     Axis          0 1 1 0
    ⍴Cases                                ⍴Actions      ⍴Masks
 16 4                                  16 6          16 4
```

`Cases` are a matrix of enclosed strings of class letters or `*`. (The *format* function `⍕` suppresses the boxes in the display.) `Actions` are a character matrix. `Masks` are a Boolean matrix.

In a parse step, the class letters of the first four rows of the stack are tested against successive rows of `Cases`; a test is true if the entry in `Cases` is `*` or if the class letter occurs in the corresponding item in `Cases`. Thus, each item in `Cases` represents a disjunctive condition. For example, `⋄←([;` , the first item of `Cases`, means the class letter being tested must be `⋄` *or* `←` *or* `(` *or* `[` *or* `;` . The applicable row of `Cases` (if any) is the first one in which all four tests are true. For example, if the four class letters of at the top of the stack are `M A F A`, then row 0 is not applicable because `M` is not in `Cases[0;0]`; row 1 is not applicable because `A` is not in `Cases[1;1]`; row 2 *is* applicable because `M A F A` are in `Cases[2;0 1 2 3]`, respectively. The logic is rendered in APL as follows:

```
i← 1 ⍳⍨ ∧/ (Cases∊⊂,'*') ∨ (4↑Stack[;1]) ∊⍨⍤1 ⊢Cases
```

The overall form of the main expression is `∧/...∨...`, a conjunction of disjunctions, and each disjunct is itself a disjunction. The line of code closely resembles that on line 4 of Table 2 in [Iverson 1987]. The result `i` is the index of the applicable row. (`i=≢Cases` if no row is applicable.)

Once an applicable row is found, `Actions[i;]` is applied to the eligible portion of the top four rows of the stack, with eligibility indicated by `Masks[i;]` (and shown in green above); the eligible portion of the stack is then replaced by the result. If no applicable row is found (`i=≢Cases`), then the bottom row of the queue is moved to the top of the stack.

With that, the current parse step ends, and a new parse step begins. In a C implementation, the index `i` in a parse step can be determined without looping or code duplication [Rich 2020].

The names in `Actions` are interpreted as follows:

`Fn1`	monadic function (§5.3)
`Fn2`	dyadic function (§5.3)
`Op1`	monadic operator (§5.3)
`Op2`	dyadic operator (§5.3)
`T2 T3`	train (§1.3)
`Assign`	assignment `←` (§1.4)
`Paren`	parentheses (§5.2)
`I0...I5`	bracket-semicolon indexing `[;]` (§2.4)
`Axis`	`f[a]` axis operator (§3.1)

Parsing terminates when no applicable row is found and the queue has no rows left. At that point, the stack should have six rows, with row 1 being the only non-marker row whose class letter is `A`, `F`, `M`, or `D` (array, function, monadic operator, or dyadic operator), otherwise a syntax error is signalled. The result of `parse` is the expression in row 1 of the stack.

In `parse`, no expression in the argument is actually executed; instead, eligible expressions are catenated together into a longer expression. For example, if the eligible expressions were `4` and `+` and `5 6`, the result would be `(4+5 6)` rather than `9 10`. An actual parser in an APL interpreter *would* execute; one can also imagine a version of `parse` which *does* execute.

`parse` has a 2-element optional left argument: in the first element, `0` (the default) specifies a parenthesized result and `1` specifies a nested array of tokens; in the second element, `0` (the default) specifies no trace, `1` specifies tracing only the non-move parse steps, and `2` specifies tracing all parse steps.

Tracing is useful as follows. The result of `parse '(⍺+⍵)×⍺-⍵'` is `'((⍺+⍵)×(⍺-⍵))'`, from which we conclude that the subtraction is done before the multiplication. But is the addition done before or after the subtraction? The question can be answered by tracing:

```
      0 1 parse '(⍺+⍵)×⍺-⍵'
┌─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┐
│◇│(│⍺│+│⍵│)│ │×│⍺│-│⍵│
├─┼─┼─┼─┼─┼─┤ ├─┼─┼─┼─┤   2 Fn2
│◇│(│A│F│A│)│ │F│A│F│A│
└─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┘
```

A left argument of `0 1` specifies parenthesized results and tracing the non-move parse steps. The top 4 entries of the stack `F A F A` test true on row 2 of `Cases`: the dyadic function `-` is evaluated, producing `(⍺-⍵)`.

```
┌─┐ ┌─┬─┬─┬─┬─┬─┬─┐
│◇│ │(│⍺│+│⍵│)│×│(⍺-⍵)│
├─┤ ├─┼─┼─┼─┼─┼─┼─┤   2 Fn2
│◇│ │(│A│F│A│)│F│A    │
└─┘ └─┴─┴─┴─┴─┴─┴─┘
```

The top 4 entries of the stack `( A F A` test true on row 2 of `Cases`: the dyadic function `+` is evaluated, producing `(⍺+⍵)`.

```
┌─┐ ┌─┬─┬─┬─┬─┬─┬─┐
│◇│ │(│(⍺+⍵)│)│×│(⍺-⍵)│
├─┤ ├─┼─┴─┴─┼─┼─┼─┤   8 Paren
│◇│ │(│A    │)│F│A    │
└─┘ └─┴─┴─┴─┴─┴─┴─┘
```

The top 4 entries of the stack `( A ) F` test true on row 8 of `Cases`: the action `Paren` is evaluated, producing `(⍺+⍵)`.

```
┌─┬─┬─┬─┬─┬─┐
│◇│(⍺+⍵)│×│(⍺-⍵)│
├─┼─┴─┴─┼─┼─┤   2 Fn2
│◇│A    │F│A    │
└─┴─┴─┴─┴─┴─┘
```

The top 4 entries of the stack `◇ A F A` test true on row 2 of `Cases`: the dyadic function `×` is evaluated, producing `((⍺+⍵)×(⍺-⍵))`.

```
((⍺+⍵)×(⍺-⍵))
```

The result of `parse`.

The current trace specifies tracing only the non-move parse steps (the left argument of `0 1` to `parse`). Each "row" of the trace displays the transposed queue (which can be empty), the transposed stack, the index of the applicable row of the parse tables, and the (upcoming) action to be applied to an eligible portion of the first four items of the stack.

The trace shows that for `(⍺+⍵)×⍺-⍵`, the function `-` is applied first, followed by `+`, then by `×`.

The parse tables can be created by executing `ParseTables ⍬`, where:

```
ParseTables←{
 t←  '/◇←([;/F/A/*' '/◇←([;]MFA/F/F/A' '/◇←([;]MFA/A/F/A' '/◇←([;]MFA/FA/M/*'
 t←t,'/◇←([;]MFA/FA/D/FA' '/◇←(]MFA/FA/F/F' '/◇←(/AFD/AFD/*' '/◇←([;]MF/AN/←/DMFA'
 t←t,'/(/DMFA/)/*' '/]/*/*/*' '/;/I/*/*' '/;/A/I/*' '/[/I/*/*' '/[/A/I/*'
 Cases∘←↑'/'(≠⊆⊢)¨t,'/◇←([;]DMF/A/J/*' '/◇←([;]MFA/F/J/*'
 Actions∘←↑'/'(≠⊆⊢)'/Fn1/Fn1/Fn2/Op1/Op2/T3/T2/Assign/Paren/I0/I1/I2/I3/I4/I5/Axis'
 t←'/0110/0011/0111/0110/0111/0111/0110/0111/1110'
 Masks∘←'1'=↑'/'(≠⊆⊢)t,'/1000/1100/1110/1100/1110/0110/0110'  }
```

The parse tables provide a measure of the cost of a language feature. For example, trains (§1.3) require parse table rows 5 and 6; bracket-semicolon indexing (§2.4) require parse table rows 9 to 14 and the five class letters `[ ; ] I J`.

`parse` does not handle the following cases:

- strand notation (§0.3) and strand assignment (§5.3)
- non-local assignment `∘←`
- contents of a dfn (§1.2)—a dfn is treated as a single token
- `⎕` functions and variables (§6.2)
- anything not part of an APL expression: control structures, `◇`, comments, etc.
- functions not returning array results or operators not returning function results
- certain anomalous constructs (which are not used in the paper)

C.2 `parse` the Function

```
parse←{
 assert (1≥≡⍵)∧(1≥≠⍴⍵)∧(0∊⍵∊' ')∧(,' ')≡1↑0⍴⍵:
 NC←0↑ Stack←4↑1↑ Queue←((⊂,'⋄')⍪'⋄')⍪(⊢,class¨)⍪tokens ⍵
 α←0 ⋄ Bx←⊃α ⋄ Tr←⊃1↓α
 trace←{
  (1=Tr)>b←⍵<≠Actions: ⍬
  a←¯3↑⍉⍪(b∨×≠Queue)⌿(b⌿(⍳⍵)⍪' ')⍪⍵⌷Actions⍪(1↓⍴Actions)↑'Move'
  ⍬⊣⎕←(⊂disp⍉Queue)⍪(⊂disp⍉¯4↓Stack)⍪⊂a
 }
 join←{(α⍴'(')⍪(∊⍵⍪¨' 'ρ⍨¨(a∊⍨⊃∘⍬¨⍵)∧1↓0⍪⍨a∊⍨⊃¨⍵)⍪α⍴')' ⊣ a←ALP⍪'¯'''}
 action←{
  e←(×α)≠⍵[;0]
  0=Bx:   (1≥α)⌿(αα⍪⍨⊂(~αα∊'IJ')join e)            @(α⍳1)⊢⍵
  αα='I': (1≥α)⌿(αα⍪⍨⊂(¯1↓e)⍪⊃⍬e)                  @(α⍳1)⊢⍵
  αα='J': (1≥α)⌿(αα⍪⍨⊂,¨'['⍪(⊂(1↓¯1↓e)⍪¯1↓⊃⍬e)⍪']')@(α⍳1)⊢⍵
          (1≥α)⌿(αα⍪⍨⊂e)                          @(α⍳1)⊢⍵
 }
 T2←{'syntax error' assert (⊂(×α)≠⍵[;1])∊' '(≠⍸⊢)' FF AD FD DA DF':
                  α('F'action)⍵}
 Op1←Op2←T3←Axis←{α('F'action)⍵}
 Fn1←Fn2←I5←     {α('A'action)⍵}
 I0←I1←I2←       {α('I'action)⍵}
 I3←I4←          {α('J'action)⍵}
 Assign←         {α( c action)⍵ ⊣ NC⍬←NC⍪⍨⍵[α⍳1;0]⍪c←⍵[α⍳3;1]}
 Paren←          {(~2|α)≠⍵}
 Move←{
  q←,¯1↑Queue
  Queue⍬←¯1↓Queue
  (q[0]≡⊂,'⍬')∧Stack[0;0]≡⊂,'.': ⍉⍪q[0]⍪'F'
  (q[1]≡'N')  ≤Stack[0;1]≡'←':   ⍉⍪q
  c←{nc⊃q}⍣(c='?')⊢c←(NC[;1]⍪'?')[NC[;0]⍳q[0]]
  'value error' assert '?'≠c:
  ⍉⍪q[0]⍪c
 }

 t←{
  i←1⍳⍨∧/(Cases∊⊂,'*')∨(4↑Stack[;1])∊¨⍎¨1⊢Cases
  t←trace⍣(×Tr)⊢i
  (0=≠Queue)∧i=≠Cases: ⍬
  i=≠Cases: ∇ Stack⍬←(Move ⍬)⍪Stack
  s←⍎'((⊢×+⍀)Masks[i;])'⍪Actions[i;]⍪' 4↑Stack'
  ∇ Stack⍬←s⍪4↓Stack
 }⍬
 'syntax error' assert (Stack[1;1]∊'DMFA')∧'⋄⋄⋄⋄⋄'≡1↓1⊖Stack[;1]:
 ⊃Stack[1;0]
}
```

C.3 Auxiliary Data and Functions

In addition to the parse tables, the following functions and variable are required.

```
)copy dfns disp
```

Get `disp` from the dfns workspace [Scholes 2019d] for creating boxed displays.

```
ALP←'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz∆⍙_0123456789'
```

The permissible characters in user names (identifiers).

```
assert←{⍺←'assertion failure' ⋄ 0∊⍵:⍺ ⎕signal 8 ⋄ shy←0}
```

As described in §9.5.

```
class←{
 assert(⍵∊'⋄⍝⎕∇')≤≠⍀⍵='''':
 (⊃⍵)∊'←()[;]':              ⊃⍵
 (⊃⍵)∊¯10↓ALP:               'N'
 (⊃⍵)∊'¯0123456789''⍬∞⍺⍵':  'A'
 (⊃⍵)∊'{':                   'DMF'[('⍵⍵' '⍺⍺'(∨⌿⍷)¨⊂⍵)⍳1]
   ⍵ ∊'⍤⍣.∘⍢⍫@⌺⍠@':          'D'
   ⍵ ∊'⌿/⍀\¨⍨⌸‖⍰&⌶':          'M'
                              'F'
}
```

Computes a class letter for the token `⍵`.

```
nc←{'AFMD?'[2 3 4 5⍳(⎕nc ⍵)+2=⊃⌽⊃⎕at ⍵]}
```

`nc` determines the class letter for a name not assigned in the APL expression provided to `parse`. Such determination is required for the following examples to work:

```
      A←2 3                   B←+⌿
      parse '*A+4'            parse '*B+4'
   (*(A+4))                (*(B(+4)))
```

```
tokens←{
 q←(⊢∨≠⍀)⍵=''''                                  ⍝ quoted strings
 a←q<⍵∊ALP                                       ⍝ alphanumerics (names)
 n←q<t∨(⍵='.')∧(1↓t⍪0)∨¯1↓0⍪t←⍵∊'0123456789¯⍬∞'  ⍝ numbers
 d←t∨×+⍀(q<⍵='{')-t←q<⍵='}'                      ⍝ dfns
 ∆←{⍵ ⍺⍺ ¯1↓0⍪⍵}
 p←(>∆ q)∨(>∆ a)∨(>∆ n)∨(>∆ d)∨(a⍱n)>q∨d∨⍵=' '
 {⍵↓⍨-⊥⍨⍵=' '}¨ ⍵ ⊂⍨ p⍀ ⍲∆ p⌿n∨q
}
```

Produces the tokens in the character string argument, individually boxed. A dfn is treated as a single token.

```
      tokens '¯1.2e¯3J¯.5E¯6 ∞ ''qi'' ⍬ {⍺+{×⍵}⍵}⍤1 2 3⊢jam'
   ┌─────────────────────────┬──────────┬─┬─────┬─┬───┐
   │¯1.2e¯3J¯.5E¯6 ∞ 'qi' ⍬│{⍺+{×⍵}⍵}│⍤│1 2 3│⊢│jam│
   └─────────────────────────┴──────────┴─┴─────┴─┴───┘
```

C.4 Tests

`tests ⍬` runs some tests (the argument `⍬` is ignored). The result is `1` if there are no problems.

```
tests←{test test1 test2 ⍬}

test←{
 assert x                          ≡ parse ⊢x←'3 ¯4.2E¯5J¯1.23E¯4':
 assert x                          ≡ parse ⊢x←'3 ''syzygy'' 4 1 5':
 assert x                          ≡ parse ⊢x←'''don''''t'' ''do'' ''it''':
 assert '(*12)'                    ≡ parse '*12':
 assert '(a⊣(a←5))'                ≡ parse 'a⊣a←5':
 assert '(((+≠)⍤1)(⊢(3 4⍴(⍳12))))' ≡ parse '+≠⍤1⊢3 4⍴⍳12':
 assert '(0 1(×⍤2 3)(⍴4))'         ≡ parse '0 1×⍤2 3⍴4':
 assert '((⍺+⍵)×(⍺-⍵))'            ≡ parse '(⍺+⍵)×⍺-⍵':
 assert '(2+(3{⍺×⍵}4))'            ≡ parse '2+3{⍺×⍵}4':
 assert '(2+({×⍵}4))'              ≡ parse '2+{×⍵}4':
 assert '((+{⍺⍺≠⍵})(⍳12))'         ≡ parse '+{⍺⍺≠⍵}⍳12':
 assert '(((+≠){⍺⍺⍤⍵⍵}4)(⍳12))'    ≡ parse '+≠{⍺⍺⍤⍵⍵}4⍳12':
 assert '((⍳4)(∘.=)(⍳5))'          ≡ parse '(⍳4)∘.=⍳5':
 assert '((*∘○)0J1)'               ≡ parse '*∘○0J1':
 t←⍎'f←g←h←+ ⋄ A←0 1 2' ⊣ t←⎕ex¨'fghA'
 assert '(f(g(h 4)))'              ≡ parse 'f g h 4':
 assert '((,[⍳2])A)'               ≡ parse ',[⍳2]A':
 assert '(2 3(,[⍳2])A)'            ≡ parse '2 3,[⍳2]A':
 assert '(((+/)[0])A)'             ≡ parse '+/[0]A':
 assert '(((+/)[(A[0])])A)'        ≡ parse '+/[A[0]]A':
 assert '((+.×)[1])'               ≡ parse '+.×[1]':
 assert '(*((A[1])×2))'            ≡ parse '*A[1]×2':
 assert '((+⍤(A[1]))(×2))'         ≡ parse '+⍤A[1]×2':
 assert '((+⍤(A[1;2]))(×2))'       ≡ parse '+⍤A[1;2]×2':
 assert '(((A[2;;1;])≠)4)'         ≡ parse 'A[2;;1;]≠4':
 assert '((4≠)(A[]))'              ≡ parse '4≠A[]':
 assert '((A[;0])(∘.×)(A[1;]))'    ≡ parse 'A[;0]∘.×A[1;]':
 assert '((A[0;0])((+.×)≠)(⍳12))'  ≡ parse 'A[0;0]+.×≠⍳12':
 p←{'(',⍵,')'}
 assert ((⊂p x),⊂p(p x),'←4')      ≡ parse¨(⊂x←'A[;]'    ),¨'' '←4':
 assert ((⊂p x),⊂p(p x),'←4')      ≡ parse¨(⊂x←'A[1;]'   ),¨'' '←4':
 assert ((⊂p x),⊂p(p x),'←4')      ≡ parse¨(⊂x←'A[;2]'   ),¨'' '←4':
 assert ((⊂p x),⊂p(p x),'←4')      ≡ parse¨(⊂x←'A[1]'    ),¨'' '←4':
 assert ((⊂p x),⊂p(p x),'←4')      ≡ parse¨(⊂x←'A[]'     ),¨'' '←4':
 assert ((⊂p x),⊂p(p x),'←4')      ≡ parse¨(⊂x←'A[1;2;3]'),¨'' '←4':
 assert ((⊂p x),⊂p(p x),'←4')      ≡ parse¨(⊂x←'A[1;;3]' ),¨'' '←4':
 t←⎕ex¨'fghA'
 1
}
```

```
Err←{0::⎕io⊃⎕dm ⋄ 0=⎕nc'α':αα ω ⋄ α αα ω}  ⍝ error trapping operator

test1←{
 A∘←1 ⊣ t←⎕ex 'A'
 assert'(A[1;;3 4;(5⍴(⍳6))])' ≡ parse 'A[1;;3 4;5⍴⍳6]':
 t←,¨'A';⊂,¨'[';(⊂(,¨'1;;');(⊂'3 4');⊂(,';');⊂(,¨'5⍴');⊂,¨'⍳6');']'
 assert t                      ≡ 2 parse 'A[1;;3 4;5⍴⍳6]':
 assert 'assertion failure'    ≡ parse Err 1 2 3:
 assert 'assertion failure'    ≡ parse Err '':
 assert 'assertion failure'    ≡ parse Err ' ':
 assert 'assertion failure'    ≡ parse Err '    ':
 assert 'assertion failure'    ≡ parse Err,¨'3+4':
 assert 'assertion failure'    ≡ parse Err 1 3⍴'3+4':
 assert 'assertion failure'    ≡ parse Err ⍬:
 assert 'assertion failure'    ≡ parse Err '⋄':
 assert 'value error'          ≡ parse Err 'asdfnotdefined':
 assert 'syntax error'         ≡ parse Err '()':
 assert 'syntax error'         ≡ parse Err '//':
 assert 'syntax error'         ≡ parse Err '\∘/':
 t←⎕ex 'A'
 1
}

test2←{  ⍝ testing trains (T2 and T3)
 t←⍎¨'f←g←h←+ ⋄ A←0 1 2' ⊣ t←⎕ex⊂'fghA'
 assert '(1∘)'                 ≡ parse '1∘':
 assert '(A∘)'                 ≡ parse 'A∘':
 assert '(0=(1+((⍟∘)0J1)))'    ≡ parse '0=1+(⍟∘)0J1':
 assert '((+⌿)⍤)'              ≡ parse '+⌿⍤':
 assert '(⍤2)'                 ≡ parse '⍤2':
 assert '(⍤A)'                 ≡ parse '⍤A':
 assert 'syntax error'         ≡ parse Err '3 A':
 assert 'syntax error'         ≡ parse Err 'A 2 3':
 assert 'syntax error'         ≡ parse Err '3×':
 assert 'syntax error'         ≡ parse Err '⍨ ⍤':

 assert '(17(+,(-,(×,÷)))2)'   ≡ parse '17 (+,-,×,÷) 2':
 assert '(4 f g)'              ≡ parse '4 f g':
 assert '((+⍤(A[0]))×-)'       ≡ parse '+⍤A[0] × -':
 assert '(+(×⍤(A[1]))-)'       ≡ parse '+ ×⍤A[1] -':
 assert '(+×(-⍤(A[2])))'       ≡ parse '+ × -⍤A[2]':
 t←⎕ex⊂'fghA'
 1
}
```

The `parse` function, the parse tables, the auxilary data and functions, and the tests, are available as a Dyalog APL workspace from http://www.jsoftware.com/papers/parse.dws (54 KB).

D *DRAMATIS PERSONÆ*

A whimsical recounting of the names which occur or can be directly inferred in the paper.

Milton Abramowitz
Phil Abrams
Wilhelm Ackermann
Al Aho
Howard Aiken
Anders Ångström
J.G. Arnold
Owen Astrachan
William Atkins
Charles Babbage
John Backus
Henry Baker
Jonathan Barman
Jacques Barzun
Jon Bentley
Bob Bernecky
Paul Berry
Sachiko Berry
Sarita Berry
George Boole
Gilad Bracha
Beverly Breed
Larry Breed
Charles Brenner
Tony Brooker
Fred Brooks
Jim Brown
Karen Brown
Adám Brudzewsky
Alex Buckley
Timothy Budd
John Bunda
William Burroughs
Anthony Camacho
Bryan Cantrill
O.G. Cassani
Hanfeng Chen
Carl Cheney
Wai-Mee Ching
Gitte Christensen
Bernard Cohen
Lothar Collatz
John Conway
Haskell Curry
Ole-Johan Dahl
John Daintree
Devil
Lee Dickey
Edsger Dijkstra
Graham Driscoll
Harold Driscoll
Dick Dunbar
Leonhard Euler
William Ewald
Adin Falkoff
Fibonacci
Roy Fielding
Oleg Finkelshteyn
Jay Foad
Daniel Friedman
Martin Gardner
Simon Garland
C.F. Gauss
Valerie Gilbert
Per Gjerløv
God
Leslie Goldsmith
Gordias
James Gosling
Alan Graham
Ronald Graham
Kate Gregory
Brent Hailpern
Hamlet
G.H. Hardy
John Harvard
Oliver Heaviside
Hans Helms
Laurie Hendren
Hercules
Israel Herstein
Carl Hewitt
William Hewlett
David Hilbert
Danny Hillis
Rob Hodgkinson
Mark Honeywell
John Hopcroft
Grace Hopper
Aaron Hsu
Michael Hughes
Roger Hui
John Iliffe
Bonnie Iverson
Derrick Iverson
Eric Iverson
Jean Iverson
Ken Iverson
Matthew Iverson
Arvid Jacobson
Stephen Jaffe
Mike Jenkins
Ronald Johnston
Bill Joy
Immanuel Kant
Joel Kaplan
Robert Kelley
Donald Knuth
Andrei Kondrashev
Pierre Kovalev
Morten Kromberg
S.E. Krueger
Phil Last
Dick Lathwell
John Lawrence
Marshall Lochbaum
Oleg Luksha
Janet Lustgarten
John McCarthy
Eugene McDonnell
Jeanne McDonnell
Jon McGrew
Donald McIntyre
Bob Metzger
George Moeckel
Roger Moore
Trenchard More
Henry Morgan
Jim Muldowney
Grant Munsey
Nick Nickolov
Johs Nielsen
Kristen Nygaard
Don Orth
David Packard
Sandra Pakin
Blaise Pascal
Oren Patashnik
Paul Penfield
Alan Perlis
Roland Pesch
Frederik Philips
Gerard Philips
Ray Polivka
Richard Potyok
Henry Rich
Spencer Rugaber
Harry Saal
Kyosuke Saigusa
Miyoko Saigusa
Jürgen Sauermann
John Scholes
Sven-Bodo Scholz
Erwin Schrödinger
Henri Schueler
G.K. Sedlmayer
Wm. Shakespeare
Jeff Shallit
Ian Sharp
Lynne Shaw
Artjoms Šinkarovs
Sisyphus
Adrian Smith
Bob Smith
Fi Smith
Howard Smith
Richard Smith
S'chn T'gai Spock
Zbigniew Stachniak
Leland Stanford Jr.
Harold Stanley
Guy Steele Jr.
Irene Stegun
David Steinbrook
Geoff Streeter
Edward Sussenguth
Roy Sykes
Alan Turing
Joey Tuttle
Jeffrey Ullman
John von Neumann
Philip Wadler
Anthony Wayne
Jim Weigang
Gregory Welch
James Wheeler
Alfred Whitehead
Arthur Whitney
Eoin Whitney
Clark Wiedmann
David Wise
Luther Woodrum
Peter Wooster
Orville Wright
Wilbur Wright
W.B. Yeats

E ALTERNATIVE HISTORIES

The following are alternative histories of APL, not in the sense of "what if" or counterfactual histories but histories with narratives organized along lines different from the present work.

- *A History of APL in 50 Functions* [Hui 2016a]
- *APL Quotations and Anecdotes* [Hui 2020a]
- *Ken Iverson Quotations and Anecdotes* [Hui 2005b]
- *Kenneth E. Iverson* [Wikipedia 2016]

F PHOTO CREDITS

Jim Brown photo by Jim Brown. Bob Bernecky photo by Bob Bernecky. *APL\360* originators photo by I.P. Sharp Associates. Gitte Christensen, John Daintree, Roger Hui, Morten Kromberg, John Scholes, and Bob Smith photos by Richard Smith.

Ken Iverson and Arthur Whitney photo by Rob Hodgkinson, used here under the CC BY-SA 4.0 license.

REFERENCES

A+. 2003. *The Online A+ Reference Manual.* aplusdev.org. http://www.aplusdev.org/refman.html (also at Internet Archive 2020-01-02 19:26:36).

Milton Abramowitz and Irene A. Stegun. 1964. *Handbook of Mathematical Functions.* National Bureau of Standards. http://people.math.sfu.ca/~cbm/aands/page_67.htm (also at Internet Archive 2018-10-18 04:33:04).

Philip S. Abrams. 1970. *An APL Machine.* Ph.D. Dissertation. Stanford University. http://www.slac.stanford.edu/pubs/slacreports/reports07/slac-r-114.pdf

ACM. 1973. Grace Murray Hopper Award. (1973). http://awards.acm.org/award_winners/breed_0694605.cfm

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

Amazon. 2018. *Alexa Presentation Language (APL) Overview.* Amazon.com, Inc. http://developer.amazon.com/docs/alexa-presentation-language/apl-overview.html (also at Internet Archive 2019-06-10 11:35:51).

Owen L. Astrachan. 2003. Bubble Sort: An Archaeological Algorithmic Analysis. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (19 Feb. 2003). http://users.cs.duke.edu/~ola/bubble/bubble.pdf (also at Internet Archive 2019-02-04 10:25:36).

John W. Backus. 1978. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM* 21, 8 (Aug. 1978). http://doi.org/10.1145/359576.359579

Henry G. Baker and Carl E. Hewitt. 1977. The Incremental Garbage Collection of Processes. *Proceedings of the Symposium on Artificial Intelligence Programming Languages, ACM SIGPLAN Notices* 12, 8 (Aug. 1977). http://doi.org/10.1145/800228.806932

Jonathan. Barman and Anthony Camacho. 1991. Panel: Is J a Dialect of APL? *Vector* 8, 2 (Oct. 1991). http://www.jsoftware.com/papers/Vector_8_2_BarmanCamacho.pdf (also at Internet Archive 2020-02-11 00:32:27).

Jacques Barzun. 2000. *From Dawn to Decadence: 500 Years of Western Cultural Life, 1500 to the Present.* HarperCollins.

Jon L. Bentley. 1983. Programming Pearls. *Commun. ACM* 26, 9 (Sept. 1983). http://doi.org/10.1145/358172.358176

Robert Bernecky. 1973. Speeding Up Dyadic Iota and Dyadic Epsilon. *Proceedings of the APL Congress 73* (22 Aug. 1973).

Robert Bernecky. 1977. System Speed-Ups. *The I.P. Sharp Newsletter, Technical Supplement* 5, 1 (Jan. 1977). http://www.snakeisland.com/IPSANewsletter_1977_01_02.pdf (also at Internet Archive 2020-02-08 06:48:11).

Robert Bernecky. 1987. An Introduction to Function Rank. *APL88 Conference Proceedings, APL Quote Quad* 18, 2 (Dec. 1987). http://doi.org/10.1145/55626.55632

Robert Bernecky. 1997. *APEX: The APL Parallel Executor.* M.Sc. Thesis. University of Toronto. http://www.snakeisland.com/ms.pdf (also at Internet Archive 2019-03-17 02:13:00).

Robert Bernecky. 2016. Zoo Story: How the SHARP APL Development Group Got Its Name. *Dyalog User Meeting 2016* (5 Oct. 2016). http://dyalog.tv/Dyalog16/?v=1N_oYD-ZkX8 (also at Internet Archive 2020-02-08 06:53:50).

Robert Bernecky, Charles H. Brenner, Stephen B. Jaffe, and George P. Moeckel. 1990. Acorn: APL to C on Real Numbers. *APL90 Conference Proceedings, APL Quote Quad* 20, 4 (July 1990). http://doi.org/10.1145/97808.97821

Robert Bernecky and Kenneth E. Iverson. 1980. Operators and Enclosed Arrays. *1980 APL Users Meeting Proceedings* (6-8 Oct. 1980). http://www.jsoftware.com/papers/opea.htm (also at Internet Archive 2019-10-02 23:07:13).

Robert Bernecky, Kenneth E. Iverson, Eugene E. McDonnell, Robert C. Metzger, and J. Henri Schueler. 1983. *SATN 45: Language Extensions of May 1983.* I.P. Sharp Associates Limited. http://www.jsoftware.com/papers/satn45.htm (also at Internet Archive 2019-10-02 23:47:25).

Paul C. Berry. 1979. *SHARP APL Reference Manual.* I.P. Sharp Associates Limited.

Frederick P. Brooks Jr. 1999. Aiken and the Harvard "Comp Lab". In *Makin' Numbers: Howard Aiken and the Computer*, I.B. Cohen and G.W. Welch (Eds.). The MIT Press, 137–142.

Frederick P. Brooks Jr. 2006. The Language, the Mind and the Man. *Vector* 22, 3 (Aug. 2006). http://archive.vector.org.uk/art10001240 (also at Internet Archive 2018-03-17 10:22:43).

James A. Brown. 1971. *A Generalization of APL.* Ph.D. Dissertation. Syracuse University. http://www.softwarepreservation.org/projects/apl/Books/AGENERALIZATIONOFAPL

James A. Brown. 1984. *The Principles of APL2, TR 03.247.* IBM Santa Teresa Laboratory, San Jose, California. http://www.softwarepreservation.org/projects/apl/Papers/PRINCIPLESOFAPL2

James A. Brown. 1988. My Favorite Idiom. (Oct. 1988). http://www.softwarepreservation.org/projects/apl/Papers/MYFAVORITEIDIOM

James A. Brown. 2016. A Personal History of APL. *APL-Journal* 35, 1-2 (2016), 3–20. http://apl-germany.de/wp-content/uploads/2018/01/APL_Journal_2016_12.pdf (also at Internet Archive 2020-02-07 19:17:27).

James A. Brown. 2017. e-mail message. (1 Nov. 2017).
This e-mail concerns the use of magic functions in APL2.

Adám Brudzewsky, Jay M. Foad, and Roger K.W. Hui. 2018. TAO Axioms. (4 Jan. 2018). http://www.jsoftware.com/papers/TAOaxioms.htm (also at Internet Archive 2020-02-07 01:23:42).

John D. Bunda. 1987. APL Function Definition Notation. *APL87 Conference Proceedings, APL Quote Quad* 17, 4 (May 1987). http://doi.org/10.1145/28315.28346

Chris Burke, Roger K.W. Hui, Kenneth E. Iverson, Eugene E. McDonnell, and Donald B. McIntyre. 1996. *J Phrases.* Iverson Software Inc. http://www.jsoftware.com/help/phrases/bond_curry.htm (also at Internet Archive 2019-10-03 07:40:58).

Bryan M. Cantrill. 2009. A Conversation with Arthur Whitney. *Queue* 7, 2 (Feb. 2009). http://doi.acm.org/10.1145/1515964.1531242

O.G. Cassani and John H. Conway. 2018. Neumbering. *The Mathematical Intelligencer* 40, 1 (March 2018). http://link.springer.com/content/pdf/10.1007%2Fs00283-017-9720-3.pdf

Hanfeg Chen, Wai-Mee Ching, and Laurie Hendren. 2017. ELI-to-C Compiler: Design, Implementation, and Performance. *PLDI Array 2017* (5 July 2017). http://doi.org/10.1145/3091966.3091969

Carl M. Cheney. 1981. *APL*Plus Nested Array System Reference Manual.* STSC, Inc. http://www.sudleyplace.com/APL/Nested%20Arrays%20System.pdf (also at Internet Archive 2019-10-11 05:07:36).

Gitte Christensen. 2006. Ken Iverson in Denmark. *Vector* 22, 3 (Aug. 2006). http://archive.vector.org.uk/art10002270 (also at Internet Archive 2017-02-02 14:48:42).

Gitte Christensen. 2014. APL in the Nordic Countries. *4th History of Nordic Computing (HiNC4)* (Aug. 2014). http://hal.inria.fr/hal-01301427/document (also at Internet Archive 2020-02-07 17:41:09).

Edsger W. Dijkstra. 1982. Why Numbering Should Start at Zero. *EWD831* 11 (Aug. 1982). http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF (also at Internet Archive 2020-01-03 03:47:28).

Graham C. Driscoll and Donald L. Orth. 1986. Compiling APL: The Yorktown APL Translator. *IBM Journal of Research and Development* 30, 6 (Nov. 1986). http://www.softwarepreservation.org/projects/apl/Papers/198611_Compiling%20APL%20The%20Yorktown%20APL%20Translator.pdf

Dyalog. 2008a. Dyalog at 25. *Vector* (Sept. 2008). http://www.dyalog.com/uploads/documents/dyalog_25.pdf (also at Internet Archive 2019-09-21 02:24:44).

Dyalog. 2008b. *Release Notes Version 12.0.* Dyalog Limited. http://docs.dyalog.com/12.0/Dyalog%20APL%20Release%20Notes.v12.0.pdf (also at Internet Archive 2015-06-06 06:57:05).

Dyalog. 2014a. *Dyalog APL Experimental Functionality—Parallel Language Features, Version 14.0.* Dyalog Limited. http://docs.dyalog.com/14.0/Dyalog%20APL%20Experimental%20Functionality%20-%20Parallel%20Language%20Features.pdf (also at Internet Archive 2016-05-07 17:17:38).

Dyalog. 2014b. Dyalog APL Idioms. (2014). http://docs.dyalog.com/14.0/Dyalog%20APL%20Idioms.pdf (also at Internet Archive 2018-02-22 04:38:39).

Dyalog. 2015. *Dyalog APL Version 14.0 Release Notes.* Dyalog Limited. http://docs.dyalog.com/14.0/Dyalog%20APL%20Release%20Notes.pdf (also at Internet Archive 2016-05-07 17:14:18).

Dyalog. 2016. *Parallel Language Features, Version 16.0.* Dyalog Limited. http://docs.dyalog.com/17.0/Parallel%20Language%20Features.pdf (also at Internet Archive 2020-02-07 19:28:11).

Dyalog. 2018a. *Dyalog APL Language Reference Guide, version 17.0.* Dyalog Limited. http://docs.dyalog.com/17.0/Dyalog%20APL%20Language%20Reference%20Guide.pdf (also at Internet Archive 2020-02-07 19:29:11).

Dyalog. 2018b. *Dyalog Programming Reference Guide, version 17.0.* Dyalog Limited. http://docs.dyalog.com/17.0/Dyalog%20Programming%20Reference%20Guide.pdf (also at Internet Archive 2020-02-11 00:44:16).

Dyalog. 2019. *RIDE User Guide, version 4.2.* Dyalog Limited. http://docs.dyalog.com/17.1/RIDE%20User%20Guide.pdf (also at Internet Archive 2020-02-26 19:33:58).

William B. Ewald. 1996. *From Kant to Hilbert: A Source Book in the Foundations of Mathematics, Volume 1.* Oxford University Press. http://books.google.ca/books?id=rykSDAAAQBAJ&pg=PA313

Adin D. Falkoff. 1969. APL\360 History. *Proceedings of the APL Users Conference at S.U.N.Y. Binghamton* (11-12 July 1969). http://www.jsoftware.com/papers/apl360history.htm (also at Internet Archive 2020-02-11 18:13:07).

Adin D. Falkoff. 1982. Semicolon-Bracket Notation: A Hidden Resource in APL. *APL82 Conference Proceedings, APL Quote Quad* 13, 1 (Sept. 1982). http://doi.org/10.1145/800071.802230

Adin D. Falkoff. 1991. The IBM Family of APL Systems. *IBM Systems Journal* 30, 4 (Dec. 1991). http://pdfs.semanticscholar.org/f7c4/72cdf7f4cb57d34c08d09f6c9a5340372678.pdf (also at Internet Archive 2019-11-20 21:25:27).

Adin D. Falkoff and Kenneth E. Iverson. 1967. *The APL\360 Terminal System.* Research Report RC-1922, IBM Corporation. http://www.jsoftware.com/papers/APL360TerminalSystem.htm (also at Internet Archive 2019-10-03 04:17:47).

Adin D. Falkoff and Kenneth E. Iverson. 1968. *APL\360 User's Manual.* IBM Corporation. http://www.bitsavers.org/pdf/ibm/apl/APL_360_Users_Manual_Aug68.pdf

Adin D. Falkoff and Kenneth E. Iverson. 1973a. *APLSV User's Manual.* IBM Corporation. http://www.softwarepreservation.org/projects/apl/Papers/197300_APL%20SV%20Users%20Manual_SH20-1460.pdf

Adin D. Falkoff and Kenneth E. Iverson. 1973b. The Design of APL. *IBM Journal of Research and Development* 17, 4 (July 1973). http://www.jsoftware.com/papers/APLDesign.htm (also at Internet Archive 2019-10-03 04:08:33).

Adin D. Falkoff and Kenneth E. Iverson. 1978. The Evolution of APL. *ACM SIGPLAN Notices* 13, 8 (Aug. 1978). http://www.jsoftware.com/papers/APLEvol.htm (also at Internet Archive 2019-10-02 22:10:44).

Adin D. Falkoff, Kenneth E. Iverson, and Edward H. Sussenguth. 1964. A Formal Description of System/360. *IBM Systems Journal* 3, 3 (1964). http://web.archive.org/web/20060813132807/http://www.research.ibm.com/journal/sj/032/falkoff.pdf

Roy T. Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation. University of California Irvine. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (also at Internet Archive 2020-02-06 12:12:35).

FinnAPL. 2008-2015. The FinnAPL Idiom Library. (2008-2015). http://aplwiki.com/FinnAplIdiomLibrary (also at Internet Archive 2019-09-26 19:26:43).

Jay M. Foad. 2017. Technical Road Map: Under the Covers. *Dyalog User Meeting 2017* (11 Sept. 2017). http://www.dyalog.com/uploads/conference/dyalog17/presentations/D03_Technical_Road_Map_Under_The_Covers.pdf (also at Internet Archive 2020-02-07 19:21:18).

Daniel P. Friedman and David S. Wise. 1976. The Impact of Applicative Programming on Multiprocessing. *Proceedings of the International Conference on Parallel Processing* (1976).

Martin Gardner. 1970. Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game "Life". *Scientific American* 223, 4 (Oct. 1970), 120–123. http://web.stanford.edu/class/sts145/Library/life.pdf (also at Internet Archive 2019-12-14 01:49:29).

C.F. Gauss. 1831. Anzeige von Theoria residuorum biquadraticorum, commentatio secunda (Notice on the Theory of Biquadratic Residues, second treatise). *Göttingische gelehrte Anzeigen* (23 April 1831). http://www.deutschestextarchiv.de/book/view/gauss_theoria_1831?p=6 Translated in [Ewald 1996].

Leslie H. Goldsmith. 1980. Corporate Communications Using the SHARP APL Mailbox. *1980 APL Users Meeting Proceedings* (Oct. 1980).

Leslie H. Goldsmith. 2010. 666 BOX. *in R.K.W. Hui, APL Quotations and Anecdotes* (2010). http://www.jsoftware.com/papers/APLQA.htm#666box (also at Internet Archive 2020-02-11 00:46:05).

James A. Gosling, William N. Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. 2015. *The Java® Language Specification, Java SE8 Edition.* Oracle America, Inc. http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (also at Internet Archive 2019-06-19 20:08:03).

Alan Graham. 1989. APL0: A Simple Modern APL. *APL89 Conference Proceedings, APL Quote Quad* 19, 4 (Aug. 1989). http://doi.org/10.1145/75144.75169

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1989. *Concrete Mathematics.* Addison-Wesley.

Kate Gregory. 2003. Managed, Unmanaged, Native: What Kind of Code Is This? (28 April 2003). http://www.developer.com/net/cplus/article.php/2197621/Managed-Unmanaged-Native-What-Kind-of-Code-Is-This.htm (also at Internet Archive 2016-03-01 20:19:30).

G.H. Hardy. 1940. *A Mathematician's Apology.* Cambridge University Press. http://www.math.ualberta.ca/mss/misc/A%20Mathematician's%20Apology.pdf (also at Internet Archive 2019-12-14 08:23:27).

Israel N. Herstein. 1975. *Topics in Algebra* (2 ed.). Xerox College Publishing.

W. Daniel Hillis. 1985. *The Connection Machine*. MIT Press. http://books.google.ca/books?id=xg_yaoC6CNEC&pg=PA46&lpg=PA46&dq=Connection+Machine+generalized+beta

Rob D. Hodgkinson. 2017. J Programming Forum post. (19 Oct. 2017). http://jsoftware.com/pipermail/programming/2017-October/049330.html (also at Internet Archive 2020-02-05 23:14:42).

Aaron W. Hsu. 2019. *A Data Parallel Compiler Hosted on a GPU*. Ph.D. Dissertation. Indiana University. http://scholarworks.iu.edu/dspace/bitstream/handle/2022/24749/Hsu%20Dissertation.pdf

Roger K.W. Hui. 1979. Algorithm 135. Generating the Group Table of S(n). *APL Quote Quad* 10, 1 (Sept. 1979). http://doi.org/10.1145/602312.602317

Roger K.W. Hui. 1981. The N Queens Problem. *APL Quote Quad* 11, 3 (March 1981). http://www.jsoftware.com/papers/nqueens.htm (also at Internet Archive 2019-10-02 23:07:56).

Roger K.W. Hui. 1987. Some Uses of { and }. *APL87 Conference Proceedings, APL Quote Quad* 17, 4 (May 1987). http://www.jsoftware.com/papers/from.htm (also at Internet Archive 2019-10-02 22:37:59).

Roger K.W. Hui. 1992a. *An Implementation of J*. Iverson Software Inc. http://www.jsoftware.com/books/pdf/aioj.pdf (also at Internet Archive 2019-08-07 13:19:08).

Roger K.W. Hui. 1992b. Three Combinatoric Puzzles. *Vector* 9, 2 (Oct. 1992). http://www.jsoftware.com/papers/Vector_9_2_Hui.pdf (also at Internet Archive 2020-02-11 00:34:11).

Roger K.W. Hui. 1993. An Implementation of J. *Vector* 9, 4 (April 1993). http://archive.vector.org.uk/art10003790 (also at Internet Archive 2017-02-22 20:26:10).

Roger K.W. Hui. 1995. Rank and Uniformity. *APL95 Conference Proceedings, APL Quote Quad* 25, 4 (June 1995). http://www.jsoftware.com/papers/rank.htm (also at Internet Archive 2019-10-02 23:20:02).

Roger K.W. Hui. 1996. J Implementation Status. (6 Jan. 1996). http://www.jsoftware.com/help/release/status.htm (also at Internet Archive 2018-10-03 13:19:55).

Roger K.W. Hui. 1998. Sparse Arrays in J. (Nov. 1998). http://www.jsoftware.com/papers/sparse.htm (also at Internet Archive 2019-10-02 22:26:17).

Roger K.W. Hui. 2000. assert. Implemented. *Jsoftware Release Notes* (21 Nov. 2000). http://www.jsoftware.com/docs/help602/release/assert.htm (also at Internet Archive 2019-04-03 13:17:44).

Roger K.W. Hui. 2004. Remembering Ken Iverson. (Nov. 2004). http://keiapl.org/rhui/remember.htm (also at Internet Archive 2019-03-31 12:17:04).

Roger K.W. Hui. 2005a. Interval Index Implemented, J 6.01 Release Notes. (29 July 2005). http://www.jsoftware.com/docs/help602/release/binsearch.htm (also at Internet Archive 2019-04-03 15:16:41).

Roger K.W. Hui (Ed.). 2005b. Ken Iverson Quotations and Anecdotes. (30 Sept. 2005). http://www.jsoftware.com/papers/KEIQA.htm (also at Internet Archive 2019-10-02 22:16:57).

Roger K.W. Hui. 2005c. NVV, in Ken Iverson Quotations and Anecdotes. (March 2005). http://www.jsoftware.com/papers/KEIQA.htm#NVV (also at Internet Archive 2019-10-02 22:16:57).

Roger K.W. Hui. 2005d. Symmetries of the Square. *J Wiki Essay* (7 Nov. 2005). http://code.jsoftware.com/wiki/Essays/Symmetries_of_the_Square (also at Internet Archive 2018-03-12 11:04:08).

Roger K.W. Hui. 2005e. Under. *J Wiki Essay* (12 Oct. 2005). http://code.jsoftware.com/wiki/Essays/Under (also at Internet Archive 2019-02-03 04:35:15).

Roger K.W. Hui. 2006. Sorting versus Grading. *J Wiki Essay* (23 Nov. 2006). http://code.jsoftware.com/wiki/Essays/Sorting_versus_Grading (also at Internet Archive 2018-03-11 14:45:10).

Roger K.W. Hui. 2007. Key. *J Wiki Essay* (9 April 2007). http://code.jsoftware.com/wiki/Essays/Key (also at Internet Archive 2018-03-12 11:00:20).

Roger K.W. Hui. 2009. Inner Product—An Old/New Problem. *British APL Association Conference 2009* (8 June 2009). http://www.jsoftware.com/papers/innerproduct (also at Internet Archive 2019-07-03 01:18:47).

Roger K.W. Hui. 2010a. Bring Something Beautiful. *Vector* 24, 4 (Dec. 2010). http://archive.vector.org.uk/art10500390 (also at Internet Archive 2018-11-29 13:22:50).

Roger K.W. Hui. 2010b. English Grammar. *Jwiki Essay* (2 Nov. 2010). http://code.jsoftware.com/wiki/Essays/English_Grammar (also at Internet Archive 2018-03-12 10:58:20).

Roger K.W. Hui. 2010c. Euler's Identity. *Jwiki Essay* (4 Feb. 2010). http://code.jsoftware.com/wiki/Essays/Euler's_Identity (also at Internet Archive 2019-10-02 22:24:35).

Roger K.W. Hui. 2010d. Hashing for Tolerant Index-Of. *APL 2010 LPA Conference Proceedings* (13-16 Sept. 2010). http://www.jsoftware.com/papers/Hashing.htm (also at Internet Archive 2019-10-02 23:43:18).

Roger K.W. Hui. 2010e. Is Origin 0 a Hindrance? (26 July 2010). http://www.jsoftware.com/papers/indexorigin.htm (also at Internet Archive 2019-10-02 23:42:56).

Roger K.W. Hui. 2010f. On Average. *Vector* 24, 2&3 (Aug. 2010). http://archive.vector.org.uk/art10500270 (also at Internet Archive 2018-04-30 02:07:44).

Roger K.W. Hui (Ed.). 2010-2020a. APL Quotations and Anecdotes. (2010-2020). http://www.jsoftware.com/papers/APLQA.htm (also at Internet Archive 2020-02-11 00:46:05).

Roger K.W. Hui. 2012. What is an Array? *Vector* 25, 3 (March 2012). http://archive.vector.org.uk/art10500690 (also at Internet Archive 2018-04-30 02:11:23).

Roger K.W. Hui. 2013a. My Favorite APL Symbol. *Vector* 26, 1 (Sept. 2013). http://archive.vector.org.uk/art10501040 (also at Internet Archive 2018-11-06 14:55:45).

Roger K.W. Hui. 2013b. Primitive Performance. *Dyalog User Conference 2013* (22 Oct. 2013). http://www.dyalog.com/uploads/conference/dyalog13/presentations/D11_Primitive_Performance (also at Internet Archive 2015-04-11 03:12:47).

Roger K.W. Hui. 2013c. Rank & Friends. *Dyalog User Conference 2013* (22 Oct. 2013). http://www.dyalog.com/uploads/conference/dyalog13/presentations/D08_Rank_and_Friends (also at Internet Archive 2018-04-30 02:07:52).

Roger K.W. Hui. 2014. Index-Of, A 30-Year Quest. *J Conference 2014* (25 July 2014). http://www.jsoftware.com/papers/indexof (also at Internet Archive 2017-11-25 20:36:02).

Roger K.W. Hui. 2015a. In Praise of Magic Functions. *Dyalog Blog* (22 June 2015). http://www.dyalog.com/blog/2015/06/in-praise-of-magic-functions-part-one/ (also at Internet Archive 2019-12-08 04:53:52).

Roger K.W. Hui. 2015b. Permutations. *Dyalog Blog* (20 July 2015). http://www.dyalog.com/blog/2015/07/permutations/ (also at Internet Archive 2018-04-30 02:10:31).

Roger K.W. Hui. 2016a. A History of APL in 50 Functions. (27 Nov. 2016). http://www.jsoftware.com/papers/50 (also at Internet Archive 2019-11-22 19:43:44).

Roger K.W. Hui. 2016b. Sixteen APL Amuse-Bouches. *Vector* 26, 4 (March 2016). http://archive.vector.org.uk/art10501480 (also at Internet Archive 2019-07-14 07:39:00).

Roger K.W. Hui. 2016c. Some Exercises in APL Language Design. (27 Nov. 2016). http://www.jsoftware.com/papers/APLDesignExercises.htm (also at Internet Archive 2018-04-30 02:13:06).

Roger K.W. Hui. 2016d. Three New Primitives. *Dyalog User Meeting 2016* (10 Oct. 2016). http://www.dyalog.com/uploads/conference/dyalog16/presentations/D06_New_Primitives_RH.zip (also at Internet Archive 2019-09-22 02:56:15).

Roger K.W. Hui. 2017a. Calculation v Look-Up. *Dyalog Blog* (13 April 2017). http://www.dyalog.com/blog/2017/04/calculation-v-look-up/ (also at Internet Archive 2018-08-17 08:02:13).

Roger K.W. Hui. 2017b. Index-Of on Multiple Floats. *Dyalog User Meeting 2017* (11 Sept. 2017). http://www.dyalog.com/uploads/conference/dyalog17/presentations/D07_IndexOf_on_Multiple_Floats.zip (also at Internet Archive 2020-02-06 17:08:44).

Roger K.W. Hui. 2017c. Stencil Lives. *Dyalog Blog* (31 July 2017). http://www.dyalog.com/blog/2017/07/stencil-lives/ (also at Internet Archive 2020-02-06 17:16:37).

Roger K.W. Hui. 2017d. ⍤ and ⍠ in Depth, SP1 Workshop. *Dyalog User Meeting 2017* (10 Sept. 2017). http://www.dyalog.com/uploads/conference/dyalog17/workshops/SP1_Version_16_Language_Features_in_Depth.zip (also at Internet Archive 2020-02-06 05:52:57).

Roger K.W. Hui. 2018a. Dyadic Grade. *Dyalog Blog* (25 April 2018). http://www.dyalog.com/blog/2018/04/dyadic-grade/ (also at Internet Archive 2020-02-06 17:20:05).

Roger K.W. Hui. 2018b. Inverted Tables. *Dyalog User Meeting 2018* (1 Nov. 2018). http://www.dyalog.com/uploads/conference/dyalog18/presentations/D14_Inverted_Tables.zip (also at Internet Archive 2020-02-06 17:55:53).

Roger K.W. Hui. 2020b. Reflections on Key. *Dyalog APL Chat Forum* (29 April 2020). http://forums.dyalog.com/viewtopic.php?f=30&t=1632 (also at Internet Archive 2020-04-30 16:03:28).

Roger K.W. Hui. 2020c. Toward Improvements to Stencil. *Dyalog APL Chat Forum* (23 April 2020). http://forums.dyalog.com/viewtopic.php?f=30&t=1625 (also at Internet Archive 2020-04-30 16:29:21).

Roger K.W. Hui and Kenneth E. Iverson. 1989-2004. *J Introduction and Dictionary.* Jsoftware Inc. http://www.jsoftware.com/help/dictionary/contents.htm (also at Internet Archive 2019-10-03 03:46:28).

Roger K.W. Hui, Kenneth E. Iverson, and Eugene E. McDonnell. 1991. Tacit Definition. *APL91 Conference Proceedings, APL Quote Quad* 21, 4 (Aug. 1991). http://www.jsoftware.com/papers/TacitDefn.htm (also at Internet Archive 2019-10-02 23:15:00).

Roger K.W. Hui, Kenneth E. Iverson, Eugene E. McDonnell, and Arthur T. Whitney. 1990. APL\? *APL90 Conference Proceedings, APL Quote Quad* 20, 4 (July 1990). http://www.jsoftware.com/papers/J1990.htm (also at Internet Archive 2019-10-02 23:14:11).

Roger K.W. Hui and Morten J. Kromberg. 2020. APL Since 1978. *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020). http://doi.org/10.1145/3386319

IBM. 1975. *APL Language GC26-3847.* IBM Corporation. http://www.bitsavers.org/pdf/ibm/apl/GC26-3847-0_APL_Language_Mar75.pdf

IBM. 1994. *APL2 Programming: Language Reference SH21-1061-01* (2 ed.). IBM Corporation. http://www.ibm.com/downloads/cas/ZOKMYKOY (also at Internet Archive 2020-02-08 05:43:15).

John K. Iliffe. 1961. The Use of The Genie System in Numerical Calculations. *Annual Review in Automatic Programming* 2 (1961), 1–28. http://doi.org/10.1016/S0066-4138(61)80002-5

Intel. 2019. Intel Intrinsics Guide – Bit Manipulation. (2019). http://software.intel.com/sites/landingpage/IntrinsicsGuide/ (also at Internet Archive 2020-01-16 17:29:52).

IPSA. 1975. APLSTAT. *The I.P. Sharp Newsletter* (April 1975), 6–7. http://www.snakeisland.com/IPSANewsletter_1975_04_05.pdf (also at Internet Archive 2020-02-04 21:07:38).

IPSA. 1978. (lead article). *The I.P. Sharp Newsletter* 6, 1 (Jan. 1978). http://www.snakeisland.com/IPSANewsletter_1978_01_02.pdf (also at Internet Archive 2020-02-05 00:08:41).

IPSA. 1979. A Decade of APL! *The I.P. Sharp Newsletter* 7, 5 (Sept. 1979). http://www.snakeisland.com/IPSANewsletter_1979_09_10.pdf (also at Internet Archive 2020-02-05 00:10:30).

IPSA. 1980. Dr. Kenneth E. Iverson. *The I.P. Sharp Newsletter* 8, 1 (Jan. 1980). http://www.snakeisland.com/IPSANewsletter_1980_01_02.pdf (also at Internet Archive 2019-08-08 16:38:48).

ISO/IEC. 1993. *Universal Coded Character Set (UCS), ISO/IEC 10646.* ISO/IEC.

ISO/IEC. 2001. *Programming Language Extended APL, ISO/IEC 13751:2001(E).* ISO/IEC.

ISO/IEC. 2011. *Information technology – Database languages – SQL Technical Reports, ISO/IEC TR 19075.* ISO/IEC. http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

Eric B. Iverson. 1978a. The Commercializing of APL\360 and Some Current Plans for Its Future. *1978 APL Users Meeting Proceedings* (Sept. 1978), 13–21.

Eric B. Iverson. 1982. Current Developments in SHARP APL. *1982 APL Users Meeting Proceedings* (1982), 83–90.

Eric B. Iverson. 2016. e-mail message. (12 April 2016).
This e-mail confirms Hui's recollection of events in 1981.

Kenneth E. Iverson. 1954a. Graduate Instruction and Research. *Proceedings of the First Conference on Training Personnel for the Computing Machine Field* (June 1954). http://www.jsoftware.com/papers/GradIR.htm (also at Internet Archive 2019-10-03 04:14:51).

Kenneth E. Iverson. 1954b. *Machine Solutions of Linear Differential Equations.* Ph.D. Dissertation. Harvard University. http://www.jsoftware.com/papers/MSLDE.htm (also at Internet Archive 2019-10-02 20:58:51).

Kenneth E. Iverson. 1962. *A Programming Language.* Wiley. http://www.jsoftware.com/papers/APL.htm (also at Internet Archive 2019-10-02 19:23:15).

Kenneth E. Iverson. 1964. Formalism in Programming Languages. *Commun. ACM* 7, 2 (Feb. 1964). http://www.jsoftware.com/papers/FPL.htm (also at Internet Archive 2019-10-02 22:17:45).

Kenneth E. Iverson. 1966. *Elementary Functions: An Algorithmic Treatment.* Science Research Associates, Inc. http://www.jsoftware.com/papers/EvalOrder.htm (also at Internet Archive 2019-10-02 22:29:54).

Kenneth E. Iverson. 1969. *The Use of APL in Teaching.* IBM Corporation. http://www.softwarepreservation.org/projects/apl/Papers/196912_The%20Use%20of%20APL%20In%20Teaching_320-0996-0.pdf

Kenneth E. Iverson. 1972a. *Algebra: An Algorithmic Treatment.* Addison-Wesley. http://www.jsoftware.com/books/pdf/algebra.pdf (also at Internet Archive 2019-10-03 04:28:56).

Kenneth E. Iverson. 1972b. *Introducing APL to Teachers.* IBM Corporation. http://www.softwarepreservation.org/projects/apl/Papers/197207_Introducing%20APL%20To%20Teachers_320-3014.pdf

Kenneth E. Iverson. 1973. *An Introduction to APL for Scientists and Engineers.* IBM Corporation. http://www.softwarepreservation.org/projects/apl/Papers/197303_An%20Introduction%20To%20APL%20For%20Scientists%20And%20Engineers.pdf

Kenneth E. Iverson. 1974. *Formal Function Definition, in Elementary Functions, Chapter 10.* IBM Corporation. http://www.jsoftware.com/papers/DirectDef.htm (also at Internet Archive 2019-10-03 04:26:27).

Kenneth E. Iverson. 1976. *Elementary Analysis.* APL Press.

Kenneth E. Iverson. 1978b. Operators and Functions. *Research Report Number #RC7091* (26 April 1978). http://www.jsoftware.com/papers/opfns.htm (also at Internet Archive 2019-10-02 22:34:40).

Kenneth E. Iverson. 1978c. Programming Style in APL. *1978 APL Users Meeting Proceedings* (18 Sept. 1978). http://www.jsoftware.com/papers/APLStyle.htm (also at Internet Archive 2019-10-02 23:03:29).

Kenneth E. Iverson. 1980. Notation As a Tool of Thought. *Commun. ACM* 23, 8 (Aug. 1980). http://www.jsoftware.com/papers/tot.htm (also at Internet Archive 2020-01-25 11:46:29). Also available at http://dl.acm.org/doi/pdf/10.1145/358896.358899.

Kenneth E. Iverson. 1981. *SATN 41: Composition and Enclosure.* I.P. Sharp Associates Limited. http://www.jsoftware.com/papers/satn41.htm (also at Internet Archive 2019-10-02 23:46:42).

Kenneth E. Iverson. 1983a. APL Syntax and Semantics. *APL83 Conference Proceedings, APL Quote Quad* 13, 3 (March 1983). http://www.jsoftware.com/papers/APLSyntaxSemantics.htm (also at Internet Archive 2019-10-02 22:16:34).

Kenneth E. Iverson. 1983b. *Rationalized APL.* I.P. Sharp Associates Limited. http://www.jsoftware.com/papers/RationalizedAPL.htm (also at Internet Archive 2019-10-02 22:35:24).

Kenneth E. Iverson. 1986a. *Applied Mathematics for Programmers.* I.P. Sharp Associates Limited.

Kenneth E. Iverson. 1986b. *Mathematics and Programming.* I.P. Sharp Associates Limited.

Kenneth E. Iverson. 1987. A Dictionary of APL. *APL Quote Quad* 18, 1 (Sept. 1987). http://www.jsoftware.com/papers/APLDictionary.htm (also at Internet Archive 2019-10-02 22:11:27). Also available at http://doi.org/10.1145/36983.36984.

Kenneth E. Iverson. 1988. A Commentary on APL Development. *APL Quote Quad* 19, 1 (Sept. 1988). http://www.jsoftware.com/papers/commentary.htm (also at Internet Archive 2019-10-02 23:13:27).

Kenneth E. Iverson. 1989-2004. personal communication. (1989-2004).
Iverson and Hui conversed in many working and informal settings, for much of which written records do not exist.

Kenneth E. Iverson. 1990. A Dictionary of J. *Vector* 7, 2 (Oct. 1990).
This version of the J Dictionary has been superseded by [Hui and Iverson 2004].

Kenneth E. Iverson. 1991a. A Personal View of APL. *IBM Systems Journal* 30, 4 (Dec. 1991). http://www.jsoftware.com/papers/APLPersonalView.htm (also at Internet Archive 2019-10-02 23:15:21).

Kenneth E. Iverson. 1991b. *Programming in J.* Iverson Software Inc.

Kenneth E. Iverson and Eugene E. McDonnell. 1989. Phrasal Forms. *APL89 Conference Proceedings, APL Quote Quad* 19, 4 (Aug. 1989). http://www.jsoftware.com/papers/fork.htm (also at Internet Archive 2019-10-02 23:13:49).

Kenneth E. Iverson and Donald B. McIntyre. 2008. Kenneth E. Iverson. (2008). http://www.jsoftware.com/papers/autobio.htm (also at Internet Archive 2020-01-25 23:17:58).

Kenneth E. Iverson, Roland H. Pesch, and J. Henri Schueler. 1984. An Operator Calculus. *APL84 Conference Proceedings, APL Quote Quad* 14, 4 (June 1984). http://www.jsoftware.com/papers/APLOperatorCalculus.htm (also at Internet Archive 2017-10-12 00:23:33).

Kenneth E. Iverson and Arthur T. Whitney. 1982. Practical Uses of a Model of APL. *APL82 Conference Proceedings, APL Quote Quad* 13, 1 (Sept. 1982). http://www.jsoftware.com/papers/APLModel.htm (also at Internet Archive 2019-10-02 22:35:02).

Kenneth E. Iverson and Peter K. Wooster. 1981. A Function Definition Operator. *APL81 Conference Proceedings, APL Quote Quad* 12, 1 (Sept. 1981). http://doi.org/10.1145/390007.805349

Michael A. Jenkins. 1989. Q'Nial: A Portable Interpreter for the Nested Interactive Array Language Nial. *Software Practice & Experience* 19, 2 (Feb. 1989). http://doi.org/10.1002/spe.4380190203

Ronald L. Johnston. 1977. APLGOL: Structured Programming Facilities for APL. *Hewlett-Packard Journal* (July 1977). http://hparchive.com/Journals/HPJ-1977-07.pdf (also at Internet Archive 2019-10-03 04:44:57).

Jsoftware. 2017. Jd Index. (2017). http://code.jsoftware.com/wiki/Jd/Index (also at Internet Archive 2018-10-25 00:29:48).

Jsoftware. 2018. Git Repositories. (2018). http://code.jsoftware.com/wiki/Git_Repositories (also at Internet Archive 2019-08-14 15:05:13).

Robert A. Kelley. 1972. APLGOL, A Structured Programming Language for APL, Report Number 320-3299. *IBM Palo Alto Scientific Center* (Aug. 1972).

Robert A. Kelley. 1973. APLGOL, An Experimental Structured Programming Language. *IBM Journal of Research and Development* 17, 1 (Jan. 1973).

Donald E. Knuth. 1968. *The Art of Computer Programming, Volume 1, Fundamental Algorithms.* Addison-Wesley.

Donald E. Knuth. 1973. *The Art of Computer Programming, Volume 3, Sorting and Searching.* Addison-Wesley.

Donald E. Knuth. 1974. Computer Programming as an Art. *Commun. ACM* 17, 12 (Dec. 1974). http://doi.org/10.1145/361604.361612

Donald E. Knuth. 1992. Two Notes on Notation. *Amer. Math. Monthly* 99, 5 (1 May 1992). http://arxiv.org/PS_cache/math/pdf/9205/9205211v1.pdf

Andrei V. Kondrashev and Oleg Luksha. 1991. A History of APL in the USSR. *APL Quote Quad* 22, 2 (Dec. 1991). http://doi.org/10.1145/130647.130656

Morten J. Kromberg. 2007. Arrays of Objects. *Proceedings of the 2007 Symposium on Dynamic Languages* (22 Oct. 2007). http://www.dyalog.com/uploads/documents/Papers/Arrays%20of%20Objects.pdf (also at Internet Archive 2019-03-31 23:53:12).

Phil Last. 2010. The fk Operator. *comp.lang.apl News Group* (26 Aug. 2010). http://groups.google.com/group/comp.lang.apl/browse_thread/thread/8b5151085e9d277d/018e033fb8f32f10#018e033fb8f32f10

Marshall W. Lochbaum. 2017. Moving Bits Faster in Dyalog 16.0. *Dyalog User Meeting 2017* (12 Sept. 2017). http://www.dyalog.com/uploads/conference/dyalog17/presentations/D08_Moving_Bits_Faster_in_Dyalog_16.zip (also at Internet Archive 2020-02-08 04:24:20).

John McCarthy et al. 1959. *LISP Programmer's Manual.*

Eugene E. McDonnell. 1976. Zero Divided by Zero. *APL76 Conference Proceedings* (22 Sept. 1976). http://www.jsoftware.com/papers/eem/0div0.htm (also at Internet Archive 2019-07-30 16:32:41).

Eugene E. McDonnell. 1977. The Story of ○. *APL Quote Quad* 8, 2 (Dec. 1977). http://www.jsoftware.com/papers/eem/storyofo.htm (also at Internet Archive 2019-07-03 01:48:11).

Eugene E. McDonnell. 1981a. *SATN 40: Complex Numbers.* I.P. Sharp Associates Limited. http://www.jsoftware.com/papers/satn40.htm (also at Internet Archive 2019-07-03 02:26:34).

Eugene E. McDonnell (Ed.). 1981b. *A Source Book in APL, Introduction.* APL Press. http://code.jsoftware.com/wiki/Doc/A_Source_Book_in_APL (also at Internet Archive 2019-10-02 22:20:59).

Eugene E. McDonnell. 1986. A Perfect Square Root Routine. *APL86 Conference Proceedings, APL Quote Quad* 16, 4 (10 July 1986). http://www.jsoftware.com/papers/eem/sqrt.htm (also at Internet Archive 2019-07-30 19:32:14).

Eugene E. McDonnell. 2003. The Magical Matrix. *Vector* 20, 2 (Oct. 2003). http://code.jsoftware.com/wiki/Doc/Articles/Play202 (also at Internet Archive 2019-07-03 09:47:30).

Eugene E. McDonnell and Jeffrey O. Shallit. 1980. Extending APL to Infinity. *APL80 Conference Proceedings* (1980). http://www.jsoftware.com/papers/eem/infinity.htm (also at Internet Archive 2019-07-30 19:32:05).

Jon McGrew. 2016. Forgotten APL Influences. *APL-Journal* 35, 1-2 (2016), 21–54. http://apl-germany.de/wp-content/uploads/2018/01/APL_Journal_2016_12.pdf (also at Internet Archive 2020-02-07 19:17:27).

Microsoft. 2017. Overview of the .NET Framework. (29 March 2017). http://docs.microsoft.com/en-us/dotnet/framework/get-started/overview (also at Internet Archive 2019-08-20 22:54:26).

Roger D. Moore. 2005. IPSANET Documents. (2005). http://www.rogerdmoore.ca/INF (also at Internet Archive 2019-07-08 08:39:17).

Roger D. Moore. 2017. e-mail message. (1 Nov. 2017).
This e-mail concerns the use of magic functions in APL.

Grant J. Munsey. 1977. APL Data: Virtual Workspaces and Shared Storage. *Hewlett-Packard Journal* (July 1977). http://hparchive.com/Journals/HPJ-1977-07.pdf (also at Internet Archive 2020-01-03 04:46:41).

Nikolay G. Nickolov. 2013. Compiling APL to JavaScript. *Vector* 26, 1 (Sept. 2013). http://archive.vector.org.uk/art10501160 (also at Internet Archive 2020-01-14 22:02:41).

Kristen Nygaard and Ole-Johan Dahl. 1978. The Development of the SIMULA Languages. *ACM SIGPLAN Notices* 13, 8 (Aug. 1978). http://doi.org/10.1145/960118.808391

Donald L. Orth. 1981. A Comparison of the IPSA and STSC Implementations of Operators and General Arrays. *APL Quote Quad* 12, 2 (Dec. 1981). http://doi.org/10.1145/586656.586662

Donald L. Orth. 2006. *Kdb+ Database Reference Manual.* Kx Systems Inc. http://legaldocumentation.kx.com/q/d/kdb+1.htm (also at Internet Archive 2020-02-08 16:36:41).

Sandra Pakin. 1972. *APL\360 Reference Manual, 2nd Edition.* Science Research Associates, Inc.

Paul L. Penfield. 1975. APL Symbols. *APL Quote Quad* 6, 1 (1975). http://doi.org/10.1145/585923.585930

Paul L. Penfield. 1979. Proposal for a Complex APL. *APL79 Conference Proceedings, APL Quote Quad* 9, 4 (June 1979). http://doi.org/10.1145/800136.804438

Paul L. Penfield. 1981. Principal Values and Branch Cuts in Complex APL. *APL81 Conference Proceedings, APL Quote Quad* 12, 1 (Sept. 1981). http://doi.org/10.1145/800142.805368

Alan J. Perlis. 1978. Almost Perfect Artifacts Improve only in Small Ways: APL is more French than English. *APL78 Conference* (29 March 1978). http://www.jsoftware.com/papers/perlis78.htm (also at Internet Archive 2019-10-02 22:19:54).

Alan J. Perlis and Spencer Rugaber. 1977. The APL Idiom List. (April 1977). http://cpsc.yale.edu/sites/default/files/files/tr87.pdf

Roland H. Pesch. 1981. Indexing and Indexed Replacement in APL. *APL81 Conference Proceedings, APL Quote Quad* 12, 1 (Sept. 1981). http://doi.org/10.1145/800142.805370

Roland H. Pesch. 2004. e-mail message. (11 Nov. 2004).
This e-mail and [Whitney 2004] confirm Hui's recollection re Whitney's invention of the rank operator in 1982.

Richard L. Potyok. 1987. Network Shared Variable Processor. *APL Quote Quad* 18, 2 (Dec. 1987). http://www.rogerdmoore.ca/INF/87QQPotyok4.htm (also at Internet Archive 2019-04-04 05:31:43). Also available at http://doi.org/10.1145/55626.55663.

Henry H. Rich. 2020. e-mail messages. (7-9 Feb. 2020).
These e-mails concern an efficient implementation of the parse model.

Kyosuke Saigusa. 1994. Use of APL in Japan. *APL94 Conference Proceedings, APL Quote Quad* 25, 1 (Sept. 1994). http://doi.org/10.1145/190271.190305

Jürgen Sauermann. 2013-2018. GNU APL. (2013-2018). http://www.gnu.org/software/apl/ (also at Internet Archive 2019-12-22 20:09:13).

John Scholes. 1996. Direct Functions in Dyalog APL. *Vector* 13, 2 (Oct. 1996). http://www.dyalog.com/uploads/documents/Papers/dfns.pdf (also at Internet Archive 2017-03-06 12:02:17).

John Scholes. 1998-2019a. D-Functions Workspace. (1998-2019). http://dfns.dyalog.com/sindx.htm (also at Internet Archive 2019-09-21 03:16:23).

John Scholes. 1998-2019b. D-Functions Workspace, at. (1998-2019). http://dfns.dyalog.com/n_at.htm (also at Internet Archive 2016-10-12 14:35:33).

John Scholes. 1998-2019c. D-Functions Workspace, cmpx. (1998-2019). http://dfns.dyalog.com/n_cmpx.htm (also at Internet Archive 2016-08-10 12:21:17).

John Scholes. 1998-2019d. D-Functions Workspace, disp. (1998-2019). http://dfns.dyalog.com/n_disp.htm (also at Internet Archive 2016-08-10 12:15:43).

John Scholes. 1998-2019e. D-Functions Workspace, parse. (1998-2019). http://dfns.dyalog.com/n_parse.htm (also at Internet Archive 2016-08-10 12:32:34).

John Scholes. 1998-2019f. D-Functions Workspace, wsreq. (1998-2019). http://dfns.dyalog.com/n_wsreq.htm (also at Internet Archive 2016-03-31 00:52:12).

John Scholes. 2013. Train Spotting in Version 14.0. *Dyalog User Meeting 2013* (22 Oct. 2013). http://www.dyalog.com/uploads/conference/dyalog13/presentations/D09_Train_Spotting_in_Version_14.pdf (also at Internet Archive 2019-09-22 02:56:16).

John Scholes. 2018. Dfns—Past, Present and Future. *Dyalog User Meeting 2018* (31 Oct. 2018). http://www.dyalog.com/uploads/conference/dyalog18/presentations/D10_Dfns_Past_Present_Future.pdf (also at Internet Archive 2020-02-08 05:06:02)., Video recording of presentation available at http://www.youtube.com/watch?v=y33XDD6ANt0.

SciPy.org. 2017. Broadcasting. *NumPy User Guide, NumPy v1.13 Manual* (10 June 2017). http://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html (also at Internet Archive 2019-05-29 07:52:14).

Jeffrey O. Shallit. 1981. Infinite Arrays and Diagonalization. *APL81 Conference Proceedings, APL Quote Quad* 12, 1 (Sept. 1981). http://doi.org/10.1145/800142.805375

Lynne C. Shaw (Ed.). 1992. *APL92 Conference Proceedings, APL Quote Quad*. Vol. 23, 1. http://dl.acm.org/doi/proceedings/10.1145/144045

Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. 2019. Convolutional Neural Networks in APL. *ARRAY 2019: Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (June 2019). http://doi.org/10.1145/3315454.3329960

SmartArrays. 1999. (1999). http://www.smartarrays.com (also at Internet Archive 2019-11-09 04:10:29).

Adrian Smith. 2004. A Strategy for Compiling APL for the .Net Framework. *Vector* 20, 4 (April 2004). http://www.jsoftware.com/papers/Vector_20_4_Smith.pdf (also at Internet Archive 2020-02-11 00:35:48).

Bob Smith. 2006-2020. NARS2000. (2006-2020). http://www.nars2000.org/ (also at Internet Archive 2020-01-23 00:03:53).

Bob Smith. 2018. e-mail messages. (18-22 Nov. 2018).
These e-mails concern Smith's APL activities between 1970 and 2018, in particular the origins of NARS2000.

Bob Smith. 2019. e-mail message. (23 Feb. 2019).
This e-mail concerns the reasons for the symbol used to denote *tally* in NARS2000.

Howard J. Smith Jr. 1979. Sorting – A New/Old Problem. *APL79 Conference Proceedings, APL Quote Quad* 9, 4 (June 1979). http://doi.org/10.1145/800136.804449

Zbigniew Stachniak. 2011. *Inventing the PC: The MCM/70 Story*. McGill-Queen's University Press.

David H. Steinbrook. 1986. *SAX Reference*. I.P. Sharp Associates.

STSC. 1983. *APL*PLUS/PC System User's Guide*. STSC, Inc.

Roy A. Sykes Jr. 2019. e-mail message. (11 Aug. 2019).
This e-mail concerns use of the STSC APL compiler.

Philip L. Wadler et al. 1989. Special Issue on Functional Programming. *Comput. J.* 32, 2 (1 Jan. 1989). http://doi.org/10.1093/comjnl/32.2.97

WaybackMachine. 2009. Snapshot of www.sigapl.org. (21 Feb. 2009). http://web.archive.org/web/20081222042714/http://www.sigapl.org/

Jim Weigang. 1985. An Introduction to STSC's APL Compiler. *APL85 Conference Proceedings, APL Quote Quad* 15, 4 (May 1985). http://doi.org/10.1145/17701.255676

Arthur T. Whitney. 1989. A. *APL89 plenary session* (Aug. 1989).
There are no known records of this presentation.

Arthur T. Whitney. 1992. personal communication, Kiln Farm. (24 May 1992).
Whitney suggested prefix instead of suffix agreement in a meeting with Hui and Iverson.

Arthur T. Whitney. 1993. K. *Vector* 10, 1 (July 1993). http://archive.vector.org.uk/art10010830 (also at Internet Archive 2020-01-17 01:20:05).

Arthur T. Whitney. 2004. e-mail message. (11 Nov. 2004).
This e-mail and [Pesch 2004] confirm Hui's recollection re Whitney's invention of the rank operator in 1982.

Arthur T. Whitney. 2006a. Abridged Kdb+ Database Manual. (13 Oct. 2006). http://legaldocumentation.kx.com/q/d/kdb+.htm (also at Internet Archive 2020-02-08 16:44:28).

Arthur T. Whitney. 2006b. Memories of Ken. *Vector* 22, 3 (Aug. 2006). http://archive.vector.org.uk/art10010840 (also at Internet Archive 2018-02-24 11:34:58).

Arthur T. Whitney. 2009. Abridged Q Language Manual. (16 March 2009). http://legaldocumentation.kx.com/q/d/q.htm (also at Internet Archive 2020-02-08 17:01:44).
Arthur T. Whitney. 2016. e-mail message. (13 April 2016).
This e-mail confirms Hui's recollection of events in 1981.
Wikipedia. 2016. Kenneth E. Iverson. (2016). http://en.wikipedia.org/wiki/Kenneth_E._Iverson
Wikipedia. 2019a. Direct function. (2019). http://en.wikipedia.org/wiki/Direct_function
Wikipedia. 2019b. John M. Scholes. (2019). http://en.wikipedia.org/wiki/John_M._Scholes
Peter K. Wooster. 1980. *SATN 35: Extended Upgrade and Downgrade.* I.P. Sharp Associates Limited. http://www.jsoftware.com/papers/SATN.pdf (also at Internet Archive 2020-02-07 17:05:07).

The "Internet Archive" hyperlink for an URL derived by entering that URL into http://web.archive.org/, resulting in http://web.archive.org/web/yyyymmddhhmmss/thatURL.

Written in honor of the centenary of the birth of Kenneth E. Iverson.