Contents

1 Falkoff and Iverson-1968: APL 1 3
2 IBM-1994: APL2 8
   2.1 APL2 Highlights . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   2.2 APL2 Primitive Functions . . . . . . . . . . . . . . . . . . . . . . 12
   2.3 APL2 Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
   2.4 APL2 Concurrency Features . . . . . . . . . . . . . . . . . . . . . 19
3 Dyalog-2008: Dyalog APL 20
4 Iverson-1987: Iverson’s Dictionary for APL 21
5 APL Implementations 21
   5.1 Saal and Weiss-1975: Static Analysis of APL1 Programs . . . . . 21
   5.2 Breed and Lathwell-1968: Implementation of the Original APL/360 25
   5.3 Falkoff and Orth-1979: A BNF Grammar for APL1 . . . . . . . . 26
   5.4 Girardo and Rollin-1987: Parsing APL with Yacc . . . . . . . . . 27
   5.5 Tavera and other-1987, 1998: IL . . . . . . . . . . . . . . . . . . 27
   5.6 Brown-1995: Rationale for APL2 Syntax . . . . . . . . . . . . . . 29
   5.7 Abrams-1970: An APL Machine . . . . . . . . . . . . . . . . . . . 30
   5.8 Guibas and Wyatt-1978: Optimizing Interpretation . . . . . . . . 31
   5.9 Weiss and Saal-1981: APL Syntax Analysis . . . . . . . . . . . . . 32
   5.10 Weigang-1985: STSC’s APL Compiler . . . . . . . . . . . . . . . . 33
   5.11 Ching-1986: APL/370 Compiler . . . . . . . . . . . . . . . . . . . 33
   5.12 Ching and other-1989: APL370 Prototype Compiler . . . . . . . . 34
   5.13 Driscoll and Orth-1986: Compile APL2 to FORTRAN . . . . . . 35
   5.14 Greleck-1999: Compile APL to Single Assignment C . . . . . . . 36
6 Imbed APL in Other Languages 36
   6.1 Burchfield and Lipovaca-2002: APL Arrays in Java . . . . . . . 36
7 Extensions to APL 37
   7.1 Brown and Others-2000: Object-Oriented APL . . . . . . . . . . 37
8  **APL on Parallel Computers**  

8.1 Willhoft-1991: Most APL2 Primitives Can Be Parallelized . . . 38  
8.2 Bernecky-1993: APL Can Be Parallelized . . . . . . . . . . . . . 41  
8.3 Bernecky-1997: APEX, An APL Compiler . . . . . . . . . . . . . 45  
8.3.1 Approach and Results . . . . . . . . . . . . . . . . . . . . 46  
8.3.2 Achieving “Portable Parallelism” with APL . . . . . . . . . . . 47  
8.3.3 Why APL Interpreters are Slow . . . . . . . . . . . . . . . . 48  
8.3.4 APL Language Features That Complicate Compilation . . . 49  
8.3.5 Speeding Up by Recognizing Arrays in Special Form . . . 49  
8.4 Ching and Ju-1991: APL on RP3 . . . . . . . . . . . . . . . . . . 50  
8.5 Greenlaw and Snyder-1991: APL on SIMD Machines . . . . . . . 53  
8.6 Ching and Katz-1994: APL on MIMD Machines . . . . . . . . . . 54  
8.7 Ju and others-1994: Loop Fusion of Array Language Primitives . 57  
8.7.1 Array Primitives . . . . . . . . . . . . . . . . . . . . . . . 58  
8.7.2 Optimization Results . . . . . . . . . . . . . . . . . . . . . 59  

9  **APL’s Market success**  

9.1 Wikipedia on APL . . . . . . . . . . . . . . . . . . . . . . . . . 59  
9.2 Wikipedia’s Criticism of APL . . . . . . . . . . . . . . . . . . . 61  
9.3 Brown-2000: What’s Wrong with APL2 . . . . . . . . . . . . . . 63
A brief overview of APL is in order, as we are about to spend a lot of time understanding it. The language was invented by Ken Iverson [Wik09b], and arose as a notation system for describing matrix computation when Iverson was an assistant professor at Harvard. When tenure was not forthcoming at Harvard, he left for IBM in 1960 and started to turn his notation system into a programming language, working with Adin Falkoff. Like LISP, it was a scripting language, but unlike LISP, it was inherently interactive. The resulting language became popular in the late 1960’s and 1970’s. Iverson became an IBM Fellow in 1970. In 1980, he left for I.P. Sharp Associates, a provider of APL in Canada. (Iverson was Canadian.) He retired in 1987 and started work on his successor to APL called J. One of the collaborators in designing J was Arthur T. Whitney [HIMW90], who went on to Morgan Stanley, where he created the languages A and A+, and then to kx systems which created the k and q languages.

1 Falkoff and Iverson-1968: APL1

In keeping with the naming convention established by IBM when it introduced the second version of APL and called it “APL2”, we call the first version of APL (APL \360) “APL1.”

In describing APL1, we follow IBM’s APL\360 user’s manual written by Falkoff and Iverson in 1968 [FI68] as well as the ISO standard published in 1983 [ISO83]. The state of the art in language design has advanced more than a little since 1968, so some of the attributes seem quaint by today’s standards, though interesting from an historic perspective. Here are the salient features of APL1.

Character Set In order to accommodate Iverson’s Greek-like characters used to denote built-in primitive functions, a non-standard keyboard was needed. IBM’s approach was to have the APL-unique characters appear as the shifted characters on specially prepared keyboards. The normally lower-case characters are used for the alphabetic characters. Possibly in keeping with the precedent set by COBOL, the alphabetic characters are always printed in upper case. Possibly in keeping with the tradition of using italic characters to denote variables in typeset published code, the upper case alphabet characters were printed in italics.

Workspace A workspace is the context of an active session. The workspace contains a program’s definitions, the value of its variables and defined functions (functions were not first-class objects in APL 1), and the current execution state. Workspaces are meant to be saved and loaded. An inactive workspace is stored in a library and a library can be shared amongst users. File systems were primitive so that libraries were identified by number. File space was limited, so that public libraries were defined as those with a number of less than 1000. Names in a workspace are in effect bound to a namespace that is coincident to the workspace.
Statements The language is statement oriented and uses a conditional goto statement (called a branch statement) for control flow decisions. Because of the richness of the functionality in the language, control decisions often can be avoided by writing a sequence of statements that imbed the conditional logic. The three types of statements are a goto, an assignment, and an expression. If an expression is entered at the interactive prompt, its value is displayed. There was also a built-in text editor that could be used to enter and change user-defined functions.

Atomic data types Numbers and characters are the atomic data types.

Numbers All numbers are conceptualized as floating point and can be written with or without a decimal point and with or without an exponent (denoted with a prefix ”E”). Negative numbers have a prefixed minus sign, which is a special symbol (a raised minus).

Characters A character is entered by typing a single quote, the character, and another single quote. The only escaped character is the quote itself, which is entered as a quote followed by another quote.

Arrays are rectangular and single-level (not nested as they were in APL2). A single element is specified by giving its indices. The number of indices is the array’s rank. A scalar, for which no indices may be used, has rank 0; a vector, rank 1; a matrix, rank 2. An empty array is an array with no components and can be created with any rank.

Array Indexing is performed by appending brackets. For a vector \( V \), \( V[I] \) denotes another array (possible a scalar or a vector) the i-th component of which is \( V_i \). For a vector \( M \), \( M[I;J] \) denotes another array (possible a scalar or a vector or a matrix) the i,j-th component of which is \( M[I,j,j] \). If I and J are vectors, then \( M[I;J] \) is a matrix, if both I and J are scalars, then \( M[I;J] \) is a scalar; otherwise \( M[I;J] \) is a vector. The form \( M[I;] \) selects all the columns and \( M[;J] \) selects all rows. The principle illustrated for matrices is extended to higher dimensions. The program may specify that the first index of each dimension is 0 or 1.

Vector Notation A vector of constants can be entered by “typing the constant components in order, separated by one or more space.” This easy-of-entry feature is called vector notation in later versions of APL.

Array Assignment is either to the entire array \( A \leftarrow expr \) or to the specified elements of an array, as in \( A[1 2 3] \leftarrow 42 57 89 \), which assigns to three different elements of \( A \).

Constant Character Vectors are denoted with a single quote, then more than one character, and finally a closing quote.

Numeric precision In the ISO standard, numeric precision is defined by the implementation (this was before the IEEE floating point standard) and
are the elementary operations on floating point numbers: addition, subtraction, multiplication, and division. What happens on underflow and overflow is likewise implementation defined. In IBM’s APL\360, at least two internal representations were used. Integers less than $2^{52}$ were stored with full precision. Arrays with range in $\{0, 1\}$ were probably stored in a compact format and were given a special name: “Boolean.” Numbers larger than $2^{52}$ and all non-integers were stored in a floating point format with “a precision of about 16 decimal digits” (hence roughly IEEE 754 double precision).

**Order of Execution** is from right to left, as the APL\360 manual states (page 3.3): “In a compound statement ... the function are executed from rightmost to leftmost ... . When parentheses are used ... the same rule applies, but, as usual, an enclosed expression must be completely evaluated before its results can be used.” This rule was introduced because APL1 had many operators and keeping track of their precedence seemed too difficult for the programmer. Moreover, the program could introduce its own function definitions and APL1 allowed them to be invoked with the same syntax as the built-in functions, so having no precedence meant that program-defined functions need not be given a precedence. The same logic would apply if APL itself introduced new functions (as APL2 did).

**Primitive Functions** are the built-in functions. They are partitioned into *scalar* and *mixed*.

**Scalar Functions** A scalar function is defined on scalar (atomic) arguments and this definition is extended to arrays in four ways: “element-by-element, reduction, inner product, and outer product.” Special characters were used to denote the scalar functions. Most of these characters were overloaded and represented both a monadic and dyadic function. The description of all of the scalar functions fit on one page of the APL\360 user’s manual, which is re-created in pertinent part in Table 1. Note: some of the character representations are approximations to the true fonts used by APL\360.

**Scalar Function Element-by-Element Extension** All the scalar functions are extended to arrays that conform, which means that the *shapes* (the vector of the array’s dimensions) are the same. The scalar functions, when extended, apply element by element. One-dimensional arrays (scalars) are expanded to a matrix of the shape of the other argument. A user-defined function (or expression) that uses only scalar functions and scalar constants is extended to arrays just as the primitive scalar functions are.

**Scalar Function Reduction** is defined by a special combination of symbols (called *operators* in later versions of APL): $f / X$, where $f$ is the scalar function. (Note that user-defined functions cannot be used in
reductions.) For a vector \( \mathbf{X} \), a reduction yields a scalar. If \( \mathbf{X} \) has dimension zero, the scalar’s value is the identify element for \( f \).
Otherwise, the scalar’s value is found by evaluating \( X_1 f X_2 f \ldots f X_n \).

For a matrix, reduction yields a vector. One must specify the dimension over which the reduction proceeds, which is the \( k \) in \( f/[k] M \). The dimension-specification is called an axis specification in later versions of APL. Reductions of matrixes of higher dimensions always yield a matrix of one lower dimension.

**Scalar Function Inner Product** is denoted \( A f.g B \), where \( f \) and \( g \) are scalar functions.

**Scalar Function Output Product** is denoted \( A ∘ . g B \). The circle symbol is a place holder in the syntax.

**Mixed Functions** are distinguished from the scalar functions in that they do not necessarily yield a scalar result nor are they extended.
element-by-element to arrays. These functions, when applied to a vector, may yield a scalar or vector. Consequently, when applying them to a higher-dimensional array, it may be necessary to specify the coordinate to act on. If \( f \) is a mixed function, then \( f[k] \) indicates that \( f \) is to be applied to the \( k \)-th coordinate. The mixed functions fit into one table, which is reproduced in pertinent part here in Table 2. In the prototypes in the table, \( A \) indicates an array, \( V \) a vector, \( M \) a matrix, and \( S \) a scalar. Some of the true APL characters are approximated: the symbols for Grade up and Grade down both contain a stroke which isn’t represented in the table. Likewise for Reverse and Rotate. The slash in the symbol for Transpose actually was slanted in the other direction.

### User-defined Functions

Functions of zero, one, and two arguments may be defined. Functions are not first-class: they cannot be assigned to a variable (later APLs relaxed this rule) and hence cannot become an item in an array. In declaring a function, one may declare variables that are local to it; other variables are global. Global variables values are found
via dynamic lookup. There is no return statement; instead the name of the function is used as its “result variable” and this name must be assigned during the function’s execution. In contrast to the then FORTRAN, all functions are recursive. Tracing of function invocations was built in. A function name is global; thus nested functions are not supported.

**Keyboard Input** from the user during an actual run is via two primitive functions. The function quad reads a line from the keyboard, evaluates it, and returns its value. The function quote-quad reads a line and returns the characters of the lines as its value.

**Dynamic Scoping** Names in APL statements were bound to values using the latest run-time definition. This means that the binding of a name can vary across statement executions. A variable is created by assigning a value to it.

## 2 IBM-1994: APL2

APL2 was offered an an enhancement to APL1. We highlight some of the salient features, based on IBM’s APL2 language reference manual [IBM94].

### 2.1 APL2 Highlights

Some of this descriptive material is redundant with the APL1 material.

**Objects in an APL2 program** are arrays, functions, and operators.

**Arrays:** all APL2 data are structured into arrays.

**Specification Arrow** ($\leftarrow$): the means for creating a variable is to assign a value to it.

**Functions** manipulate array structure of perform calculations on array data.

**Primitive Function** has a name that is a symbol.

**Primitive Operators** have a name that is a symbol and are similar to functions but apply to functions and produce a derived function. Operators provide a restricted form of first-class functions.

**Properties of Arrays** Each array has a rank which is its number of dimensions. A scalar has rank 0, a vector has rank 1, and a matrix has rank 2. If an array has rank 2 or more, its last axis is called its column axis and its next-to-last axis is called its row axis. If a matrix has rank 3, its first axis elements are called planes or pages. Each axis contains zero or more items. The shape of an array is a vector containing the number of items in each dimension. If the array has one or more axes
that have zero items, the array is *empty*. An empty array necessarily has
at least rank 1. All arrays are rectangular.

An item in an array may be an array. If so, the array has a *depth* of
greater than 1.

**Numeric Data** is always complex (has an imaginary part). Subtypes of
Boolean (zero or one) and integers are acknowledged (and presumably
there storage was optimized). Complex numbers are entered in one of
three forms: real and imaginary parts separated by a “J” and no spaces
(the number is real if the imaginary part is zero), magnitude and angle
in degrees separated by a “D” and no spaces (the number is real if the
angle is an integral multiple of 180), and magnitude and angle in radians
separated by a “R” and no spaces (the number is real if the angle is an
integral multiple of pi).

**Character Data** are distinct from numeric data and are entered between
single quote characters.

**Vector Notation** The juxtaposition of two or more arrays in an expression
results in a vector whose items are the arrays. This notation extends
what is possible in APL1, as the juxtaposed arrays could be the result of
expressions as well as constants.

**Name and Symbol Binding** Introduction of nested arrays introduced
complexity in parsing statements. There is a hierarchy of binding
strength (in descending order):

1. **brackets**: binds to what is on their left. Brackets indicate indexing
   if the object to their left is an array; brackets indicate axis
   specification if the object to their left is a function or operator.

2. **specification left**: left arrow binds to what is on its left. Thus in
   APL, A ⍵←C results in ⍵ being assigned the value of expression C
   and then vector notation being applied to join the value of A and
   the value of C. The entire expression to the right of the arrow is an
   array that is evaluated before the assignment is made.

3. **right operand**: dyadic operator binds to its right operand. In APL,
   +.x/ is interpreted as (+.x)/. There is no binding between
   operators.

4. **vector**: an array binds to an array. Vector binding (the result of
   using vector notation) is stronger than the binding of a function to
   its arguments. Thus in APL 1 2 + 3 4 means (1 2)+(3 4) and 1
   2 3/A means (1 2 3)/A.

5. **left operand**: an operator binds to its left operand

6. **left argument**: a function binds to its left argument
7. right argument: a function binds to its right argument. Left argument binding is stronger than right argument binding. This means that the evaluation of an expression begins with the rightmost function whose arguments are available.

8. specification right: a left arrow binds to what is on its right

Name and Symbol Binding, Once Again IBM published in 2002 [IBM02] an APL2 Language Summary which gives (page 28 and following) another explanation of when the right-to-left order is not followed. The exceptions to the right-to-left rule, in order of priority as listed as:

1. Parentheses: always permitted, provided the contained expression yields an array, a function, or an operator.
2. Bracket notation: have an implied function or operator which is on the left of the left bracket and are tightly bound to that object. Brackets are used for array indexing and to specify the axis for some functions and operators.
3. Specification object, which is the l-value to the right of the assignment arrow. The assignment arrow behaves like a function and returns the value of its right argument.
4. Vector notation: a series of value expressions separated by spaces is treated as a vector. The items within the vector notation are evaluated from right to left.
5. Operand binding, which applies to operators. Operators are functions that take an function as an argument and produce another function. Most operators are monadic; monadic operators have their operand on their left. When a monadic operator is found, the APL2 interpreter looks to its left for an operand. There are three cases: the left operand is an array, then vector notation is applied and the resulting array is the operand; the left operand is a function which is the operand; the left operand is another operator, in which case APL2 produces a derived operator from the left operand and the derived operator becomes the argument for the original operator. A dyadic operator has left and right operands, which are function.

6. Normal function processing order, that is, right to left without precedence.

Specification of a variable is via the left arrow: \( A \gets \text{expression} \).

Specification of multiple variables The notation \( A_1 \gets A_2 \gets \ldots \gets A_n \) assigns the value of the expression to each array variable.

Specification of several variables from a vector The notation \( (A_1 \ A_2 \ldots \ A_n) \gets \text{expr} \) is shorthand for several statements of the form \( A_{i} \gets \text{expr}_{i} \).
Selective Specification  an an expression that selects from an array values that will be replaced. Alarming, “deviations exist on some platforms” (page 40 of [IBM94]) so even IBM had trouble getting the notation uniformly implemented. Selective expression replaces a whole array or a subset of it. The selection expression on the left works like an l-value in C; it specifies not the values (C’s r-values) but rather the locations of items in the array on the left. Once the locations are determined by evaluating the selection expression, these rules are applied:

1. If the left is a whole array, the right array replaces it.
2. If the right is a scalar or an array with empty shape when ones in its shape vector are removed, then the right is paired with each item on the left and these rules are applied recursively.
3. If the left and right have the same shape (when ones in the shape are ignored), then corresponding items from the left and right are paired and these rules are applied recursively.

Primitive Functions Types  are either scalar or nonscalar. A scalar primitive function is one that indexing distributes over: If F is a monadic scalar primitive function then (F R)[I] is equivalent to F R[I]; If G is a dyadic scalar primitive function then (L G R)[I] is equivalent to L[I] G R[I].

Primitive Monadic Scalar Functions are conjugate, negative, direction, reciprocal, magnitude, floor, ceiling, exponential, natural log, Pi times, factorial, not, roll.

Primitive Dyadic Scalar Functions are add, subtract, multiply, divide, residue, minimum, maximum, power, logarithm, circular, binomial, or, and, nor, less, not greater, equal, not less, greater, not equal.

Primitive Monadic Nonscalar Functions are shape, ravel, reverse, transpose, enclose, disclose, first, interval, enlist, grade up, grade down, matrix inverse, depth, execute, format.

Primitive Dyadic Nonscalar Functions are reshapre, catenate, laminate, rotate, transpose, partition, pick, drop, take, without, index of, member, grade up, grade down, deal, find, encode, decode, matrix divide, match, format, indexing, index.

Extension of Monadic Scalar Functions These functions are defined on simple scalar arguments and extended to other arguments using these rules:

1. If the argument is a scalar, apply the definition.
2. If the argument is not empty, apply the function independently to each simple scalar of its argument. The result has rank, shape, and depth identical to that of its argument.
3. If the argument is empty, apply the related fill function to the prototype of the argument.

**Extension of Dyadic Scalar Functions** These functions are defined on simple arguments and extended to other arguments using these rules:

1. If both arguments are scalars, apply the definition.
2. If one or both arguments are empty arrays, apply the related fill functions to the empty arguments.
3. If the arguments have the same shape, apply the function to the corresponding items. The result has the same shape as the arguments.
4. If one argument is a scalar or a one-item vector, pair the scalar or one-item vector with each of the other items and apply the function. The result has the same shape as the nonscalar argument.
5. If the arguments are nested, apply these rules to the corresponding pairs.

**Fill Functions** When a primitive scalar function is presented with empty arguments or when a function derived from the operators each or array product is presented with empty arguments, the function is not executed. Instead a related fill function is executed, if one is defined. All primitive monadic and dyadic scalar functions have the same fill function: When the prototypes of the empty arguments are simple scalars, return a zero prototype. When the prototypes of the empty arguments are not simple scalars, apply the fill function to each item recursively. When one argument is a scalar and the other is empty, apply the fill function to each item recursively.

Fill functions for primitive nonscalar functions are used when the functions are derived from the operators each and array product. Details are not provided here (see the manual, page 57).

**Type and Prototype** The *type* of an array yields a zero for each number in the array and a blank for each character. The type has the same structure as the array. The *prototype* of an array is the type of its first item.

### 2.2 APL2 Primitive Functions

We list the primitive functions, provide a prototype, and provide a brief description for each. The notation is the same as from the APL2 manual:

- L: left argument
- R: right argument
- LO: left operand
• RO: right operand
• X: axis

Add: \( Z = L + R \) Add R to L. Scalar function.

Binomial: \( Z = L! R \) The number of distinct combinations of R things taken L at a time. Scalar function.

Boolean functions: not R; L and R; L or R; L nand R; L nor R. Scalar functions.

Bracket index: \( A[I] \) Selects subarrays of A according to the index arrays L. Within I, semicolons separate arrays that define positions along each axis. When a vector is indexed, I is a single index array. When a matrix is indexed, two array indices separated by a semicolon may be used. Index arrays may be elided to indicate that all indices for the corresponding axes. Index values may be repeated; the indicated item is selected repeatedly. Bracket indexing can be used for selective specification (aka, assignment).

Catenate: L, R or catenate[L, R] If L and R are nonscalar arrays, join L and R along the last axes. If L and R are scalar arrays, the result is a two-item vector. The result of a catenate applied to simple scalars or vectors is the same as a simple vector created by vector notation.

Catenate with Axis: catenate[L, R, X] Join L and R along the axis indicated by X, which is a scalar or one item vector.

Ceiling: ceiling R For real R, yield the smallest integer that is not less than R using the comparison tolerance. For complex R, what is yielded depends on the relationship between the real and imaginary parts of R.

Circle functions: sin R, arcsin R, ... For completeness with APL2, one must implement the following functions: sine, arcsine, cosine, arccos, tangent, arctan, sinh, arcsinh, cosh, arcosh, tanh, arctanh, real R, imaginary R, phase R.

Compress from Slash: LO compress R Select subarrays along the last axis under the control of the vector LO. Compress can be used for selective specification.

Compress with axis from Slash: compress[LO, R, X] Select subarrays along the X axis under control of the vector LO.

Conjugate: \( *R \) Yield R with its imaginary part negated. Scalar function.

Deal: L?R Select L integers at random from the population \{0, 1, ..., R-1\} without replacement.
Decode: \( \text{L decode R} \) Yield the values of array \( \text{R} \) evaluated in a number system with radices \( \text{L} \). When \( \text{L} \) is a scalar and \( \text{R} \) is a vector, yields the value of a polynomial evaluated at \( \text{R} \).

Depth: Reports level of nesting

Direction: \( \ast \text{R} \) Yield the number of magnitude 1 with the same phase as \( \text{R} \) for nonzero \( \text{R} \). For 0 \( \text{R} \), yield Z.

Disclose: \( \text{disclose R} \) Structure the items of \( \text{R} \) into an array, whose rightmost axes come from the axes of the items of \( \text{R} \).

Disclose with Axis: \( \text{disclose[R,X]} \) Structure the items of \( \text{R} \) into an array. X defines the axes of the result into which the items of \( \text{R} \) are structured.

Divide: \( \text{L/R} \) Divide \( \text{L} \) by \( \text{R} \)

Drop: \( \text{L drop R} \). Remove subarrays from the beginning or end of the Ith axis of \( \text{R} \), depending on whether \( \text{L[I]} \) is positive or negative.

Drop with Axis: \( \text{drop[L,R,X]} \) Remove subarrays from the beginning or end of the \( \text{X}[I] \)th axis of \( \text{R} \).

Each (dyadic): \( \text{each[L0,L,R]} \) Apply the function \( \text{L0} \) between corresponding pairs of items of \( \text{L} \) and \( \text{R} \), where \( \text{L0} \) is a dyadic function.

Each (monadic): \( \text{each[L0,R]} \) Apply the monadic function \( \text{L0} \) to each item of \( \text{R} \).

Enclose: \( \text{enclose R} \) Yield a scalar array whose only item is \( \text{R} \).

Enclose with Axis: \( \text{enclose[R,X]} \) Yield an array whose items are the contiguous subarrays along the set of axes indicated by \( \text{X} \): the set of axes indicated by \( \text{X} \) is enclosed.

Encode: \( \text{L encode R} \) Yield the representation of \( \text{R} \) in the number system whose radices are \( \text{L} \).

Enlist: \( \text{enlist R} \) Yield a simple vector whose items are the simple scalars in \( \text{R} \). Note: if \( \text{R} \) is a simple vector (not nested), then enlist and ravel are equivalent. Enlist can be used for selective specification.

Execute: \( \text{execute R} \) Evaluate the statement represented by the character vector \( \text{R} \).

Expand (from Backslash): \( \text{L0 \ R or expand[L0,R]} \) Expand the last axis of \( \text{R} \) under the control of the boolean vector \( \text{L0} \). Can be use for selective specification.

Expand with Axis (from Backslash): \( \text{expand[L0,L,R]} \) Expand the Xth axis of \( \text{R} \) under the control of the boolean vector \( \text{L0} \). Can be used for selective specification.
**Exponential:** \( \text{exp } R \) Yield the \( R \)th power of the base of the natural logarithms. Scalar function.

**Factorial:** \( !R \) For positive integer \( R \), yield the product of all positive integers through \( R \). For all numbers except negative integers, yield the value of the Gamma function of \( R+1 \).

**Find:** \( L \ \text{find } R \) Yield a boolean array that maps to \( R \).

**First:** \( \text{first } R \) Yield the first item of \( R \) in row-major order. If \( R \) is empty, yield the prototype of \( R \).

**Floor:** \( \text{floor } R \) If \( R \) is real, yield the largest integer that does not exceed \( R \). If \( R \) is complex, the value yielded depends on the relationship between \( R \)'s real and imaginary parts.

**Format:** \( \text{format } R \) Yield a simple character array whose appearance is the same as the display of \( R \).

**Format by Example:** \( L \ \text{format } R \) Transform \( R \) into a character array that is displayed according to the format model \( L \). \( L \) contains control characters.

**Format by Specification:** \( \text{fprintf}[L,R] \) or \( L \ \text{fprintf } R \) Transform \( R \) into a character array.

**Grade down:** \( \text{codegrade\_down } R \) Yield a vector of integers that puts the subarrays along the first axis of \( R \) in descending order.

**Grade down with Collating Sequence:** \( L \ \text{grade\_down } R \) : Yield a vector of integers that puts the subarrays along the first axis of \( R \) in descending order according to the collating sequence \( L \), which is a simple nonempty scalar character array.

**Grade up:** \( \text{grade\_up } R \) Yield a vector of integers that puts the subarrays along the first axis of \( R \) is ascending order.

**Grade Up with Collating Sequence:** \( L \ \text{grade\_up } R \) Yield a vector of integers that puts the subarrays along the first axis of \( R \) in ascending order according to collating sequence \( L \).

**Index:** \( L \ \text{index } R \) Yield cross-sections of \( R \) using a list of index arrays \( L \).

**Index Of:** \( L \ \text{index\_of } R \) or \( L \ \text{index\_of}[L,R] \) Yield the first occurrence in \( L \) of items in \( R \).

**Index with Axis:** \( \text{index\_of}[L,R,X] \) Yield the cross-sections of \( R \) using a list of index arrays \( L \) that correspond to axes \( X \).
**Inner Product (from Array Product):** $L \cdot_0 R \cdot_0 R$ Yield the combination of subarrays along the last axis of $L$ with subarrays along the first axis of $R$ by applying an $R_0$ outer product followed by an $L_0$-reduction of each item of that result.

**Interval:** $\text{interval } R$ Yield $R$ consecutive ascending integers beginning with 0.

**Laminate:** $\text{laminate}[L,R,X]$ Yield the join of $L$ and $R$ by forming a new axis of length 2 which is filled with $L$ and $R$, where $X$ is a simple scalar fraction.

**Logarithm:** $\text{log}(R,b)$ Yield the base “$b$” logarithm of $R$. A scalar function.

**Magnitude:** $|R|$ or $\text{magnitude } R$ Yield the distance between 0 and $R$. A scalar function.

**Match:** $L \text{ match } R$ Yield 1 if $L$ and $R$ have the same structure and data; otherwise, yield 0.

**Matrix Divide:** $L \text{ matrix divide } R$ Yield the solution of a system of linear equations according to the values and shapes of $L$ and $R$. Both $L$ and $R$ must be simple numeric arrays of rank 2 or less.

**Matrix Inverse:** $\text{matrix_inverse } R$ Yield the inverse of $R$, which must be nonsingular and a simple numeric array of rank 2 or less.

**Maximum:** $L \text{ max } R$ Yield the larger of $L$ and $R$, where both are real numerics. A scalar function.

**Member:** $L \text{ member } R$ Yield a boolean array with the same shape of $L$ such that an item of the result is 1 if the corresponding element of $L$ can be found anywhere in $R$ and is 0 otherwise.

**Min:** $L \text{ min } R$ Yield the smaller of $L$ and $R$. A scalar function.

**Multiply:** $L \cdot R$ Yield the product of $L$ and $R$.

**Natural Logarithm:** $\text{log } R$ Yield the logarithm of $R$ to the base of the natural logarithms. A scalar function.

**Negative:** $-R$ Yield $R$ with its sign reversed. A scalar function.

**Outer Product (from Array Product):** $L \odot_0 R \odot_0 R$ Apply the function $R_0$ between pairs of items, one from $L$ and one from $R$, in all combinations.

**Partition:** $L \text{ partition } R$ Partition $R$ into an array of vectors specified by $L$.

**Partition with Axis:** $\text{partition}[L,R,X]$ Partition $R$ into an array of vectors specified by $L$ along axis $X$. 
Pi Times: $\pi R$ Yield pi times R. A scalar function.

Pick: $L \text{ pick } R$ Yield at item of R as specified by the path indexes L.

Power: $L \text{ power } R$ Yield L raised the Rth power. A scalar function.

Ravel: $\, R \text{ or } \text{ravel } R$ Create a vector from the items in R taken in row-major order.

Ravel with Axis: $\text{ravel}[R, X]$ Create an array that contains the items of R reshaped according to axes X.

Reciprocal: $1/R$ Yield 1 divided by R. Scalar function.

Reduce (from Slash): $L \text{ reduce } R$ has the effect of placing function LO between adjacent pairs of items along the last axis of R and evaluating the resulting expression for each subarray, where LO is a dyadic function.

Reduce N-wise (from Slash): $\text{reduce}[L, R, LO]$ L defines the number of items along the last axis to be considered in each application of the function LO to the subarrays along the last axis of R.

Reduce N-wise with Axis (from Slash): $\text{reduce}[L, R, LO, X]$ Reduce, where L defines the number of items along the Xth axis to be considered in each application of the function to the subarrays along the Xth axis.

Reduce with Axis (from Slash): $\text{reduce}[LO, R, X]$ The function LO is placed between adjacent pairs of items along the Xth axis of R.


Replicate (from Slash): $L \text{ replicate } R$ Repeat each subarray along the last axis under the control of the vector LO.

Replicate with Axis (from Slash): $\text{replicate}[LO, R, X]$ Repeats each subarray along the X axis under the control of vector LO.

Reshape: $L \text{ reshape } R$ Yields the items of R in an array of shape L.

Residue: $L|R$ For real positive numbers L and R, yields the remainder of dividing R by L. Otherwise, a generalized residue is returned (see manual page 227). Scalar function.

Reverse: $\text{reverse } R$ Yield an array with the items of R reversed along the last axis.

Reverse with Axis: $\text{reverse}[R, X]$ Yield an array with items of R reversed along the Xth axis.

Roll: $\text{?R}$ Select an integer at random from the population index R. Scalar function.
Rotate: \( L \text{ rotate} \ R \) Yield an array with the items of \( R \) rotate \(|L|\) positions along the last axis. Can be used for selective specification.

Rotate with Axis: \( \text{rotate}[L,R,X] \) Yield an array of items of \( R \) rotated \(|L|\) positions along the \( X\)th axis. Can be used for selective specification.

Scan (from Backslash): \( \text{L0 scan} \ R \) The \( I\)th item along the last axis is determined by the LO-reduction reduction of a certain form (see page 239).

Scan with Axis (from Backslash): \( \text{scan}[L0,R,X] \) The \( I\)th item along the \( X\)th axis is determined by the LO-reduction of a certain form (page 240).

Shape: \( \text{shape} \ R \) Yield the size of each axis of \( R \) as a vector.

Subtract: \( L - R \) Yield the result of subtracting \( R \) from \( L \). Scalar function.

Take: \( L \text{ take} \ R \) Yield the subarrays from the beginning or end of the \( I\)th axis of \( R \), according to whether \( L[I] \) is positive or negative. Can be used for selective specification.

Take with Axis: \( \text{take}[L,R,X] \) Yield the subarrays from the beginning or end of the \( X[I]\)th axis of \( R \), according to whether \( L[I] \) is positive or negative. Can be used for selective specification.

Transpose (General): \( L \text{ transpose} \ R \) Yield an array similar to \( R \) but with axes permuted according to \( L \), or if \( L \) included repetitions of axes, do something similar (see page 251).

Transpose (Reversed Axes): \( \text{transpose} \ R \) Yield an array similar to \( R \) but the order of the axes of \( R \) reversed. Can be used for selective specification.

Without: \( L~R \) or \( L \text{ without} \ R \) Yield the items in \( L \) that do not occur in \( R \).

### 2.3 APL2 Operators

APL2 provides these operators, listed here for ease of reference, as they are in the long list of functions in the prior section. An operator takes as input one or more functions and produces another function that is somehow derived from the input functions. Notation: \( LO \), left operator; \( RO \), right operator; \( X \) axis; \( L \), left argument; \( R \), right argument; “” meant to approximate the true APL2 symbol which is two raised dots.

- Compress: \( L/0/R \)
- Compress with Axis: \( L/[X]R \)
- Each, Deriving Dyadic: \( L \text{ L0"R} \)
- Each, Deriving Monadic: \( L0"R \)
Expand: $L \backslash R$
Expand with Axis: $L \backslash [X] R$
Inner Product: $L \bullet R$
Outer Product: $L \circ R$
Reduce: $L \backslash R$
Reduce N-wise: $L \backslash R$
Reduce N-wise with Axis: $L \backslash [X] R$
Reduce with Axis: $L \backslash [X] R$
Replicate: $L \backslash [X] R$
Replicate with Axis: $L \backslash [X] R$
Scan: $L \backslash R$
Scan with Axis: $L \backslash [X] R$

2.4 APL2 Concurrency Features

APL2 provides a way to implement concurrency through shared variables, which are variables shared between two “processors” (which can be implemented via an operating system process or thread). A processor can be an APL program in execution or an auxiliary processor (presumably another program in execution not written in APL).

The only means of communication between processors is bilateral (one process to exactly one other process) and through a shared variable. Either partner can set a value for a shared variable or use its value. The implementation guarantees that at any one time the shared variable has one value; thus setting the value of a shared variable is atomic from a concurrency perspective. A processor can share variables with many other processors.

APL2 has built-in functions to establish and maintain the shared variables:

- Determine if a variable’s sharing is in effect (if the variable is coupled)
- Offer to share variable
- Control a shared variable by setting or querying its access controls and access state
- Retract the shared variables
- Determine whether variables offered for sharing are in fact being shared
- Specify how long to wait for an offer to share a variable to be accepted
• Wait for an offer to share to be accepted

The following outlines the general approach to sharing a variable:

1. Offer to share the variable
2. Ensure that another processor accepted the offer
3. Set access controls for the variable
4. Access the variable
5. Retract the sharing of the variable

The access controls provide the means of coordinating between the processors. These controls are set in a vector, which has four components, each of which, when set to 1, implies a control:

1. If set, you cannot set the variable twice without an intervening access by your partner.
2. If set, the partner cannot set the variable twice with an intervening access by you.
3. If set, you cannot use the variable twice with an intervening set by your partner.
4. If set, the partner cannot use the variable twice with an intervening set by you.

Your processor’s execution is blocked if your process attempts to set or access a shared variable before your partner has set or accesses the variable, as determined by the access controls.

3 Dyalog-2008: Dyalog APL

Dyalog APL is a version of APLs with nested arrays [Dya08]. It has all the standard APL2 features: a workspace, right-to-left evaluation, exotic character set.

It extends APL2 (the original version from IBM) in these ways:

- **Unicode** is supported.

- **Business graphics** are built in.

- **Namespaces** Provides a way to limit the scope of a name. Used in classes, so that an object uses only a single name in the top-level namespace. Provides for static (lexical names); names are otherwise dynamically scoped. Uses the traditional dot notation to access the namespace: `namespace.name`. Namespaces are first-class, so they can be stored in arrays.
Classes Full class functionality, include methods, properties, fields, constructors, and destructors. Dyalog has focused on the Windows operating system and hence its classes can derive from .Net classes.

Ambivalence Functions may be defined with one function body, by placing the left-most argument in the function header within curly brackets.

Specification is enhanced beyond what IBM’s APL2 supports. (Most likely, some of IBM’s specification forms are not supported by Dyalog.)

Threads are directly supported. They are created with a new primitive spawn or using an asynchronous function call. The APL threads are all multi-tasked under one OS thread, so there is no real multi-threading going on. The execution of a line of a function is made an indivisible operation.

To synchronize threads, Dyalog APL provides a token which may be placed in a token pool. In addition, there are indivisible functions that put and get tokens from a pool. The idea is similar to the P and V operators from Dijkstra. The tokens and pools can implement semaphores and latches.

In addition to tokens, Dyalog APL provides a hold statement to mark a critical section.

Modern Control Structures are provided, not just APL2’s goto. The structures provided include: if-then-else, while, repeat, for, and case. The syntax requires a keyword at the beginning of a statement. In my view, the syntax is ugly: example: ;If, where the initial colon is required, probably to avoid reserving a keyword. To simplify writing a sequence of statement that use a common object, a with statement is provided.

To handle exceptions, Dyalog APL has the trap statement.

A return statement is also provided.

Triggers are mechanism that cause a function to be called whenever a variable is assigned. These are seen as implementation mechanisms for class objects properties.

Hash tables are used in implementing certain array primitives, namely: index of, membership, intersection, union, without, and matrix iota (an idiom, no a primitive). If the programmer does nothing, the hash table is constructed when the function is called and discarded at function completion. The programmer can cause the hash table to be retained, by using an operator to bind the function and the array and then calling the derived function. (This is a wordy solution and makes programming hard.)

Tail recursion is converted to iteration but only for functions defined as dynamic functions using a special syntax.

Ken Iverson provided [Ive07a] a dictionary for APL that covered the most common built-in primitive functions. For each, he provided an English name, which we reproduce here as Table 3 and continue in Table 4. The thought process is: if we are replacing the APL special symbols (the Greek letters) with reserved words, let’s consider the reserved words created by Iverson himself.

5 APL Implementations

We review the public literature on APL implementations.

5.1 Saal and Weiss-1975: Static Analysis of APL1 Programs

Early in APL1’s history, Harry Saal and Zvi Weiss analyzed [SW75] statically 32 APL workspaces chosen to be “polished, semi-professional programs.” The application areas included plotting, text editing, PERT analysis, numerical integration, STATPAK statistical analysis, digital computer simulations (the authors were employed by IBM’s research lab in Haifa, Israel), linear programming, operations research, and digital circuit design.

Size of APL functions is smaller than FORTRAN functions, presumably because of the higher-level semantics for the APL operators. According to a study by Knuth [Knu71], FORTRAN program have on “average 100 statements (excluding continuations) per END statements [an END statement marks the end of a FORTRAN subroutine], where the APL average is about 12 lines, nearly an order of magnitude smaller.”

APL lines are semantically complex as measured by the number of nodes in the parse tree for the statement. For APL, the average parse tree had 9 nodes per statement. No corresponding data were quoted for FORTRAN statements.

APL programs contain much less documentation than do FORTRAN programs. Less than 1 percent of the APL workspace lines were comments. Knuth found that between 11 and 28 percent of the FORTRAN lines were comments. The authors attribute the difference to workspace storage constraints for APL and to APL’s richer semantics. We add that APL’s culture did not include writing extensive comments.

Statement distribution follows the 80-20 rule. (At the time of their analysis, the APL scan statement was new and hence presumably under-represented in the workspaces relative to its steady-state frequency.) A few functions dominate:

- Primitives (the built-in functions) were 73 percent
<table>
<thead>
<tr>
<th>APL symbols</th>
<th>Monadic Names</th>
<th>Dyadic Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>+      −      ×      ÷</td>
<td>mate/minus/trend/per</td>
<td>plus/num/times/per</td>
</tr>
<tr>
<td>*      ⊙</td>
<td>power/log</td>
<td>power/log</td>
</tr>
<tr>
<td>&lt;      ≥      &gt;      ≦      ≧      ≜      ¬     ∧      ∨</td>
<td>box</td>
<td>before</td>
</tr>
<tr>
<td>≡      ≳      ≽      ≼</td>
<td>cycle</td>
<td>fore</td>
</tr>
<tr>
<td>≟      ≞      ≟</td>
<td>nub in</td>
<td>equal</td>
</tr>
<tr>
<td>≠      ≞</td>
<td>nubsieve</td>
<td>unequal</td>
</tr>
<tr>
<td>≡</td>
<td>match</td>
<td></td>
</tr>
<tr>
<td>¬      ∧      ∨</td>
<td>not</td>
<td>less</td>
</tr>
<tr>
<td>∧      ∨      ...</td>
<td>and(lcm)/or(gcd)/nand/nor</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>right/left</td>
<td>right/left</td>
</tr>
<tr>
<td>⊥      ⊤      ...</td>
<td>words</td>
<td>base</td>
</tr>
<tr>
<td>...</td>
<td>execute</td>
<td>execute</td>
</tr>
<tr>
<td>...</td>
<td>format</td>
<td>format</td>
</tr>
<tr>
<td>o</td>
<td>pi</td>
<td>circle</td>
</tr>
<tr>
<td>...</td>
<td>reverse</td>
<td>rotate</td>
</tr>
<tr>
<td>...</td>
<td>upset</td>
<td>rowel</td>
</tr>
<tr>
<td>...</td>
<td>cant</td>
<td>cant</td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>residue</td>
</tr>
<tr>
<td>!</td>
<td>factorial</td>
<td>out of</td>
</tr>
<tr>
<td>⌊      ⌈      ...</td>
<td>floor/ceiling</td>
<td>maximum/minimum</td>
</tr>
<tr>
<td>⌈      ⌊      ...</td>
<td>nub/raze</td>
<td>take/drop</td>
</tr>
<tr>
<td>∈      τ      ...</td>
<td>raze in</td>
<td>in</td>
</tr>
<tr>
<td>ρ      ...</td>
<td>shape</td>
<td>reshape</td>
</tr>
<tr>
<td>?      ...</td>
<td>roll</td>
<td>deal</td>
</tr>
<tr>
<td>,      ...</td>
<td>ravel/table</td>
<td>by/over</td>
</tr>
</tbody>
</table>

Table 3: Iverson’s Dictionary of APL: Extract (Part 1)
<table>
<thead>
<tr>
<th>APL symbols</th>
<th>Monadic Names</th>
<th>Dyadic Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊃</td>
<td>box open</td>
<td>link</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mix index</td>
</tr>
<tr>
<td>{</td>
<td>all</td>
<td>from</td>
</tr>
<tr>
<td>...</td>
<td>inverse</td>
<td>inverse</td>
</tr>
<tr>
<td>...</td>
<td>copy down</td>
<td>copy down</td>
</tr>
<tr>
<td>...</td>
<td>v-down</td>
<td>alpha-way v-down</td>
</tr>
<tr>
<td>...</td>
<td>expand down</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>scan down</td>
<td></td>
</tr>
<tr>
<td>v}</td>
<td>select</td>
<td>merge</td>
</tr>
<tr>
<td>v⊂</td>
<td></td>
<td>swap</td>
</tr>
<tr>
<td>...</td>
<td>with/with</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>constant</td>
<td>constant</td>
</tr>
<tr>
<td>...</td>
<td>cut</td>
<td>cut</td>
</tr>
<tr>
<td>...</td>
<td>prefer/defer</td>
<td>prefer/defer</td>
</tr>
<tr>
<td>...</td>
<td>upon</td>
<td>upon</td>
</tr>
<tr>
<td>m.v</td>
<td></td>
<td>tie</td>
</tr>
<tr>
<td>u.n</td>
<td>ply</td>
<td></td>
</tr>
<tr>
<td>u.v</td>
<td></td>
<td>dot product</td>
</tr>
<tr>
<td>...</td>
<td>combining rank</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>union</td>
<td></td>
</tr>
<tr>
<td>u ∩ v</td>
<td>intersection</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>define</td>
<td>define</td>
</tr>
</tbody>
</table>

Table 4: Iverson’s Dictionary of APL: Extract (Part 2)
• Subscripting was 18 percent
• User function calls were 4 percent
• Reductions were 3 percent
• All others were the remained

Vectors and matrices dominate On average across the 32 workspaces,
• Vectors (rank 1) were 56 percent of arrays
• Matrices (rank 2) were 43 percent of arrays
• Three dimensional arrays (rank 3) were 1 percent of arrays
• Arrays of rank 4 or higher were not used.

APL programmers used matrices (rank 2 arrays) more than
FORTRAN programmers “in order to take advantage of the operations
available on structured data in APL.”

Goto statements were 28 percent of all function lines. We speculate that
since a goto is needed to implement looping constructs in APL, that
some type of built-in looping primitives are useful in array languages.

5.2 Breed and Lathwell-1968: Implementation of the
Original APL\360

Breed and Lathwell describe [BL68] in 1967 some of the details of the original
APL\360 implementation. Recall that this APL provided many of the facilities
that later came to be associated with the operating system, namely, support of
terminals, a file system (for workspaces). At the time of the writing, APL was
still only a research project and hence not generally available.

Speed versus compiled code was one-tenth to one-fifth.

Memory footprint Forty to fifty concurrent users were supported on a
System360 Model 50 that had about 264 KB of core. Workspaces were
swapped to disk via the APL supervisor, which did many of today’s
operating system functions.

Workspace storage was allocated in two portions. Growing down from the
top of the workspace was the execution stack. Growing from the bottom
was the heap, which held arrays, APL source statements, and
implementation-defined objects. Objects were referenced indirectly
through a fixed symbol table so that garbage collection could move
objects around.

Source statement compilation was minimal: just conversion of the source
statement to an internal form on a symbol-by-symbol basis. These
conversions were done: a function identifier was converted to a 16-bit
pointer to the symbol table; a special APL character was converted to an 8-bit code; and a constant was converted to a code, count, and binary value. This amounts to lexical analysis without any real parsing. “APL is sufficiently close to Polish prefix form, because of its right-to-left operator precedence, that further conversion would yield little improvement in processing speed. It would, moreover, complicate the conversion back to input form that is done for typewrite display.” (page 5).

**Local Variables** are found at input time and bound to the symbol table. Hence, no search is needed at run-time. [As nearly as I can tell, q still does this!]

**Syntax analysis** was through Conway transition diagrams, a form of syntax-directed interpretation.

**Numerical data types** were logical, integer, and long floating point.

**Overhead time for interpretation** “Syntax analysis and setup for evaluation of a simple expression is two to three milliseconds, while execution time for each scalar element is typically 40 to 250 microseconds.” (page 8) Let’s do some analysis. Supposed the average values held for a given instance and the array had 1000 items. Then the total execution time would be total = setup + execution = \(2.5 \times 10^{-3} + 150 \times 10^{-6} = (2500 + 150,000) \times 10^{-6}\), which might seem to indicate the the syntax analysis did not slow down the total time by much, however, there must have been execution inefficiency in the 150,000 portion relative to compiled code, since the average speed of execution was said to be one-tenth to one-fifth that of compiled code.

### 5.3 Falkoff and Orth-1979: A BNF Grammar for APL1

A.D. Falkoff, one of the authors of the original APL\360 manual, was also a co-author along with D.L. Orth of a paper with contains IBM’s internal standard for APL1 [FO79]. This paper contains the actual standard as an appendix. Part of the standard is a BNF grammar for APL1, which we quote in part below. Our notation is meant to be compatible with that of yacc.

```plaintext
statement:     label-name ':' statement-body
               | statement-body

statement-body: expression
                | RIGHT-ARROW
                | comment
                | null

expression:    subexpression
```

26
subexpression:  
  | subexpression dyadic-identifier expression  
  | monadic-identifier expression  
  | object GETS expression

simple-expression:  
  | object GETS expression

object:  
  | variable-identifier
  | variable-identifier '[ index-expression ]'
  | QUAD
  | QUAD-QUOTE

index-expression:  
  | expression
  | ' : ' index-expression
  | expression ' ; ' index-expression

niladic-identifier:  
  | NILADID-DEFINED-FUNCTION

dyadic-identifier:  
  | dyadic-function-name
  | dyadic-scalar-primitive
  | dyadic-mixed-primitive
  | dyadic-derived-function
  | dyadic-system-name

dyadic-function-name:  
  | DYADIC-DEFINED-FUNCTION

dyadic-scalar-primitive:  
  | a list was given

dyadic-mixed-primitive:  
  | a list was given

dyadic-derived-function:  
  | dyadic-scalar-primitive ' . ' dyadic-scalar-primitive
  | CIRCLE ' . ' dyadic-scalar-primitive
  | ' , ' '[ expression ]'
  | ' / ' '[ expression ]'
  | SLASH-HYPHEN '[ expression ]'
  | '\ ' '[ expression ]'
  | CIRCLE-BAR '[ expression ]'
  | CIRCLE-HYPHEN '[ expression ]'
5.4 Girardo and Rollin-1987: Parsing APL with Yacc

Jean Jacques Girardot and Florence Rollin provide in 1987 [?] a way to construct a yacc parser for APL2. After first observing that APL2 is not LR(k) by providing a counterexample, they show that if one syntactic ambiguity is eliminated by the lexical analyzer, the resulting grammar is LALR(1). The ambiguity is the use of square brackets to denote both an axis specification and indexing. There “simple solution . . . consists in having the tokenizer recognize axis brackets from index brackets [based on the fact that] an opening axis bracket always follows a primitive symbol such as [list given in the original].” Thus the lexical analysis routine first squeezes out any extra spaces on an input line and then defines compound lexemes that end with an open square bracket. These lexemes are converted into two lexical units: one for the character preceding the open square bracket and then a character that is substituted for the open square bracket.

5.5 Tavera and other-1987, 1998: IL

Tavera, Alfonseca, and co-authors in 1985 [TAR85] and again 1991 ([ASR91]) described the approach used by IBM to implement several APL and APL2 program products.

The initial goal of the IL stream of research was to speed up the implementation of APLs. This was done using the following process [TAR85]:

1. Write an APL interpreter in IL, which is a machine independent language with low-level (machine-like) semantics. The syntax of IL was a simplified form of the syntax of APL.

2. Write a compiler to translate IL programs into assembly language. This was facilitated by designing IL’s semantics to be close to machine language semantics. This compiler, which was target-machine specific, was usually written in APL or APL2.

3. Compile the IL interpreter for the target machine.

APL interpreters for System/370, Series/1, System/6000, and the IBM Personal Computer were created in this way.

A second goal was to speed up APL and this goal followed two strategies: extend the language with even more powerful primitives and implement the interpreter in a low level language (namely IL).

IL itself is interesting, as it is a low-level semantics version of APL. IL’s instructions were defined after surveying the assembly languages of various IBM systems, with the goal of being able to compile an IL instruction into a few assembly language instructions. Key attributes of IL include:
Memory model The machines memory is a vector of consecutively-numbered bytes.

IL Constants are numeric and single character literals (which represent a number in the ASCII encoding or the EBCDIC encoding).

Identifiers Only five characters may be used and identifiers beginning with the letter ‘Q’ are reserved for IL’s use. The type (and hence storage) of a variable depends on its first letter. These types are supported: one-byte integers, two-byte integers, full-word integers (which are also pointers), floating-point values, internal label (the target of a goto), a public labels in a program, and named constants.

Data structures The only one is a vector. A scalar is defined as a vector of one element.

Variable declarations must be used, though assigning an initial value serves as a declaration.

Variable equivalences are defined via variable = 1-value and create aliases.

Executable instructions Its no surprise that the language used to implement APL also is parsed right to left and that functions have no precedence.

Assignment statements assign to an entire variable, implement the equivalent of C’s += and -=, and have a special form that assigns the address of an expression to a variable (similar to an assembly language load address instruction).

Execution control is via a goto statement in one of several forms: uncondition, goto if a condition holds, and a computed goto.

Relative to C, IL was judged by the paper author to have better memory management, and C was judged to have better type-constraint capabilities.

5.6 Brown-1995: Rationale for APL2 Syntax

APL1 had a simple-to-describe syntax and APL2’s syntax is much more complicated. James Brown published an article in 1985 [Bro85] that explains the rationale for the syntax.

APL1’s syntax rules were simple:

- “All functions have equal precedence,”
- “Functions are executed from right to left.”
- “Operators have higher precedence than functions.”
APL2 generalizes some of the notions in APL1. For example, in APL1 the reduction operator worked with only certain primitive functions. In APL2, the desire was to allow reduction to work with program-defined functions. The desire to do this while still having function invocation not have an explicit notation led to considerable complexity.

We summarize the rules of APL2’s syntax (as given by Brown).

**Object classes** are arrays, functions, operators.

**Function valence** All functions are ambivalent. Which one is invoked is determined by the context.

**Operator valence** An operator is either monadic or dyadic as determined by its definition, not its context.

**Syntax classes** are

- arrays
- functions
- monadic operators
- dyadic operators
- assignment arrow
- brackets

**Parentheses rule** Parentheses are used for grouping and must be paired. What’s inside must yield an array, function, or operator. Redundant parentheses are permitted.

**Expression** a sequence of names and symbols, each from the syntax classes, punctuated by paired parentheses

**Right to left rule** In an unparenthesized expressions without operators, functions are evaluated from right to left.

**Function precedence** There is none.

**Rewrite rule for character vectors** If a vector in parentheses is made up entirely of single characters, it may be rewritten with a single pair of enclosing quotes

**Binding hierarchy** From highest to lowest:

1. *brackets* bind to what is on the left
2. *assignment left* bind to what is on its left
3. *right operand*: a dyadic operator binds to which is on its right
4. *vector* bind an array to an array (this is the basis for vector notation)
5. **left operand**: an operator binds to what is on its left
6. **left argument**: bind a function to its left argument
7. **right argument**: bind a function to its right argument
8. **assignment right**: bind a left arrow to what is on its right

**Brackets and monadic operators** have no binding strength on the right

**Right arrow** is syntactically a function

**Niladic functions** are syntactically arrays

The binding hierarchy seems impossible to remember and hence would lead to a decrease in productivity as programmers attempt to understand the code of others.

### 5.7 Abrams-1970: An APL Machine

In 1970, eight years after APL was introduced, Philip Samuel Abrams as his Ph.D. Thesis [Abr70] designed at a high level a machine appropriate for APL execution (after compilation). The goals were that the machine’s “primitive operation and data structures should include those of APL” and that the machine should “execute programs as efficiently as possible.”

To address the efficiency concern, Abrams defines two processes: beating and drag-along and beating. “Drag-along is the process of deferring evaluation of operands and operators as long as possible.” [This is a form of lazy evaluation.] For example, in the expression `take(3, 2 * -V)`, which negates V, multiplies each element by 2 and creates a new vector containing the first three elements, one need only do the arithmetic on the first three elements. The key is to not simplify the expression too soon. Another example (both examples are from Abrams) is `A+B+C+D` where only the final result needs to be determined, not the intermediate results of say `A+B` and `C+D` and then the sum of those to partial answers. Abrams’ machine deferred “all array calculations . . . as long as possible in order to gain wider context of information about the expression being calculated.” (page 192) Drag-along is a way to “minimize [probably meaning reduce] any unnecessary calculations in evaluating APL programs.”

Beating is “the transformation of code to reduce the amount of data manipulation during expression evaluation” [Ayc03]. The Abrams’ machine (which was never implemented), the selection operation on an array was evaluated by beating which amounted to “changing the storage mapping functions of its constituent array operands.” Beating is confusion, so we offer Abrams summary definition: “If arrays are represented in row-major order and if the representation of the storage access function for an array is kept separate from the array value, then the result of applying a selection operator to an array can be obtained simply by transforming the mapping function.” (page 191) [Abram’s beating is maintaining a data structure that maps indices to an array into storage locations. Rather than rebuild the storage for an array, rework the mapping structure.]
Regarding parallelism and our interest in many cores, Abrams notes (page 194) that the implied parallelism of APL that makes drag-along and beating possible was not fully exploited in his design. He envisions “multiple copies of [key evaluation algorithms] working simultaneously on different parts of an expression or program.” And so do we.

5.8 Guibas and Wyatt-1978: Optimizing Interpretation

In [GW78] Leo J. Guibas and Douglas K. Wyatt describe optimization of APL interpretation. Rather than naively interpret every statement and every expression exactly as written, Guibas and Wyatt make these observations:

- Many APL operations do not produce data that are actually assigned to a variable or returned by a function. Instead, these operations specify data elements that other functions will operate on. In the language of Guibas and Wyatt, they “rename the elements” and are called grid selectors.

- Once these grid selectors are identified in the parse tree (which is built as an early stage of the interpretation), they may be systematically moved to the leaves of the parse tree.

- Once the operations are at the leaves, a technique that today we would call just-in-time compilation can be used to produce code that executes quickly, basically by operating not on the multiple indices that the program may have specified, but instead on a single index that operates on raveled arrays (these are APL arrays in row-major order that are created by APL’s ravel operator).

- The compiled code may be saved and reused under certain conditions.

Examples of application of these ideas:

- If a reshape operator \texttt{a reshape b} is conforming, defined as \texttt{shape b} is a suffix of \texttt{a}, then inner and outer products can be eliminated and pushed down to leaves where they become “scalar operators and reductions along the last dimension.” As such, they can be compiled into a simple loop. “The compiled code is very efficient. A minimum number of loop variables is maintained and accessors [for arrays] are shared among as many expression atoms as possible.”

- If \texttt{transpose a + b} is to be computed, one approach is to compute the element-by-element sum of \texttt{a} and \texttt{b} and then transpose it. This requires building a large intermediate result. Another approach is to directly compute the transposed result without generating \texttt{a + b}. Note that the code for \texttt{a} and \texttt{b} cannot have any side effects for this re-write to work.
Another idea is *slicing*, which identifies a portion of an expression that will be used repeatedly, computing its value once and saving it, and the reusing the computed value.

Some optimizations the authors discovered operate on the expression and its subexpressions, others depend on knowing the sizes of the arrays.

5.9 Weiss and Saal-1981: APL Syntax Analysis

Zvi Weiss and Harry Saal describe an approach to compiling APL [WS81]. Optimizations that are possible were APL to be compiled are listed as:

- Conformability checking (to prevent run-time errors)
- Data type determination (citing the SETL deduction scheme as an example of what could be done)
- Idiom recognition
- Drag-along and beating
- Other optimizations in the spirit of compilers for FORTRAN, PL/1 (the article was published in 1981).

However, APL has dynamic binding for user-defined names, with the result that APL statements cannot be parsed (beyond lexical analysis) until run-time. So, Weiss and Saal describe a way to deduce the types of certain names at compile time. An analysis of APL programs in use “supports the hypothesis that the restricted APL language [a subset of APL in which all types are deducible by their algorithms] encompasses almost all existing APL code.” Their conclusion was based on the analysis of 32 workspaces which had 8,593 lines of APL code.

For APL1, all that is required for compilation is that the compiler, for user-defined names, can determine if the name is that of an array or of a function, and if for a function, the valence of the function. This is a small restriction and its not surprising that much code actually written does not take advantage of the ability to parse a statement in multiple ways depending on the dynamic binding of the names in the statement. Weiss and Saal determine data types through “a static, compile-time, inter-procedural data-flow analysis.”

5.10 Weigang-1985: STSC’s APL Compiler

Jim Weigang, then an employee of STSC (a major APL vendor), offered in 1985 an introduction to STSC’s APL compiler [Wei85]. Weigang describes what is today the conventional compilation approach: lexical analysis, parsing, data flow analysis (to determine types and shapes of arrays), code generation, and code optimization.
To analyze speedup of the compiled code relative to the interpreted code, Weigang distinguished between iterative APL programs and non-iterative program.

**Iterative programs** speedup between 8 and 200 times, because the compiler can often eliminate array allocations in the loop. More iterations lead to greater speedups; larger arrays lead to smaller speedups.

**Non-iterative programs** speedup between 0.9 (a slowdown) and 2 times. The lower speed up is caused by few opportunities for loop merging than in iterative code. Larger arrays lead to smaller speedups.

Because programs have a mix of iterative and non-iterative sections, typical speedups were 1.5 to 8 times.

### 5.11 Ching-1986: APL/370 Compiler

Wai-Mee Ching, a researcher at IBM’s Watson Research Center, published in 1986 a description of an APL compiler for the System/370 [Chi86]. There are three main results:

- Execution time of the compiled code that is close to that of IBM’s VS FORTRAN on a System/370. Substantial details are not given.

- Some compilation techniques that are needed in scalar languages were found to not be needed in compiling an array language like APL, because the array language more directly expresses the aggregate calculation. The techniques that were not used in the APL compiler were common subexpression elimination, strength reduction, and loop-invariant code motion. Some of the optimizations a scalar compiler might have used were imbedded directly in the run-time of the compiled APL.

- To generate good code, Ching’s compiler deduced where possible that an array had a special form. The special forms that were valuable to recognize were: a scalar or one-element vector, an array which dimensions and sizes completely known at compile time, a vector of unknown length, a non-vector array with a know number of dimensions but unknown sizes. The compiler deduced the attributes of arrays and where it could not, prompted the user for the information. The compiler never generated code for the general case (an array of unknown number of dimensions), because it always required a declaration to remove this case.

### 5.12 Ching and other-1989: APL370 Prototype Compiler

Wai-Mee Ching and his collaborators describe in 1989 [CNS89a] the APL370 compiler, an compiler accepting “an essential subset of APL” and compiling it
directly into System/370 assembly code. The compiled code did not use the 3090 vector facility.

The focus is on comparison of run-times of the compiled code versus the APL2 release 2 interpreter, which also did not use the 3090 vector facility. The article notes that an interpreter “is likely to outperform a compiler on the code for many/most primitives, but still be beaten by a compiler on whole examples.” The logic isn’t given, but perhaps refers to the ability of an interpreter to find special cases that their 1989-era compiler could not find. In fact the compiler was noted to still be buggy at the time of the report.

In spite of the experimental nature of the compiler, its “compiled code actually executes faster [than] FORTRAN if powerful APL features such as bit-manipulation are involved.” This means that if the APL program were written in good APL style, avoiding loops and use the APL aggregate array operators, the resulting compiled program was more efficient than the looping FORTRAN program. The result reflects in part IBM’s FORTRAN compiler’s maturity in 1989; compilers today are presumably much stronger.

Regarding what to optimize, the APL370 compiler did not implement Abrams’ drag-along and beating [Abr70] because optimization of APL primitives such as boolean compare were believed to be more important.

Compilation speeds were noted to be “much faster than other APL compilers” the authors were aware of. They note that STSC’s compiler takes 34 hours (!) to compile an electronics design application. The APL370 compiler was written entirely in APL, was probably interpreted (though this isn’t stated in the article), was said to have some obvious optimizations not implemented (like writing the generated code using blocks instead of a line at a time), and yet compiled in between 1.14 to 5.75 lines per cpu second.

The performance of the compiled code is analyzed in terms of the speedup relative to the interpreted code. Four groups of benchmark programs were tested.

**Sequential-style APL code** written in a style similar to that required by scalar languages and hence regarded as to be avoided. Speedups of between 20 times and 144 times were observed.

**Iterative or recursive code** and hence presumably slow on the interpreter. [Iteration was used to avoid the build-up of intermediate results that would be too large for an APL workspace to hold.] Speedups of between 3 and 27 times were observed.

**Good-style APL code** “already efficient on the interpreter,” where one example ran slower (its speedup was 0.84) and another had a speedup of 19. The medium speedup was about 2. The authors note that for their compiler, “a compiler which is not very efficient on computation-intensive primitives has no advantage over the interpreter on good APL style code when data arrays become large. Hence it is very important for a compiler to generate very efficient code for computationally intensive primitives.”
Selected APL primitive functions where speedups of between 1.67 and 5 were observed.

5.13 Driscoll and Orth-1986: Compile APL2 to FORTRAN

In 1986, Graham Driscoll and Donald Orth describe an early APL compiler that translates (compiles) APL into FORTRAN [DO86]. The authors says that the first APL compiler was for the Burroughs APL-700. This “compiler” kept a syntax tree and regenerated it only when necessary. (Today we would call this an interpreter, because it executed the statements directly.) Second was Hewlett-Packard’s APL-3000 which was by today’s standards a real compiler. It deduced array element storage types and the shapes (dimensions) of arrays. The HP compiler generated an high-level intermediate language rather than machine code. Other APL compilers used just-in-time compilation to “generate code at run time, based largely on dynamic information, and aim for code that is quite specific to the instance.” Driscoll notes that other efforts that were based on “one-for-one replacement of APL primitives with code in [a] target language” and did not use type inference for the arrays and yet “even [this] most straightforward approaches to APL compilation can yield dramatic improvements in many cases.” Regarding STSC’s APL compiler, Driscoll notes that its main optimization effort was around the generated scalar code, not transforming the APL aggregate operators into more efficient forms. [Thus it had to compete with mature compilers for scalar languages.] Like most other APL compilers, they restricted the language such that the rank and type of arrays was known at compile time and was unvarying. Another restriction was that the distinguished axis for an array primitive be known at compile time.

The generated FORTRAN code could be compiled and then executed from an APL workspace using an experimental extension to APL2 (an auxiliary processor, in the APL jargon) that allowed compiled code to be called. The speedups observed for the compiled code relative to interpreted code were generally between 1.8 times and 5.0 times. Generally, the larger the array, the lower the speedup, so that one’s intuition about amortizing interpretation overhead is supported. In one exception case (sorting), the compiled code achieved a speed of between 124 and 151. Matrix inversion also had a large speedup. “We have compiled and timed a number of other APL programs and found that they ran from 2 to 250 times faster after compilation.” (The sample isn’t described, so the results do not necessarily represent typical production programs.)
5.14 Grelck-1999: Compile APL to Single Assignment C

Clemens Grelck and Sven-Bodo Scholz describe [GS99] results of an effort to accelerate APL programs using SAC, Single Assignment C. Their approach is to hand translate three APL programs into equivalent SAC programs by replicating the APL operators with their roughly-equivalent SAC counterparts. (SAC is contains many of the APL primitives as built-in or standard functions [Sch07]). The three applications were: seismic signal processing using a one dimensional convolution, another signal processing application, and a mesh generation program.

The hand-translated SAC codes achieved speedups of between 2 and 500. In part these good results may have been caused by the benchmark APL system, which ran under Windows and hence may not have been highly optimized (because the Windows market may not have demanded a high-performing APL interpreter). In fact, the authors wondered what could make the APL system so slow.

6 Imbed APL in Other Languages

Rather than implement a stand-alone language, one could imbed APL in other languages or their runtimes. For example, Microsoft has imbedded OCAML into its .Net runtime.

6.1 Burchfield and Lipovaca-2002: APL Arrays in Java

Joseph Burchfield and Samir Lipovaca, both then with the New York Stock Exchange, describe [BL02] in 2002 their effort to use APL-like arrays in Java. Their motivation was the “natural” correspondence of relational database tables to array and the observation that scalar oriented languages (like C and Java) “do not permit manipulation of whole arrays in a single operation,” which destroys the correspondence between the data in memory and on disk in the table.

Their approach is:

1. Implement a class NiceKeyboard to allow “easy input from the keyboard.” This mimics the APL interactive prompt.

2. Implement a class APL as a subclass of NiceKeyboard, apparently to allow for interactive input during debugging. This class has one private instance variable which is an Java Vector. As such, the vector expands and contracts as needed. Rather than store numeric items in the vector, they seem to have decided to store characters items, inasmuch they describe how an APL output convention, that a blank line separates the planes of a 3D array, is implemented: “a \n will be inserted at the end of the corresponding vector.”
3. For the APL primitives, they implement a Java method for each, using overloading to get monadic and dyadic forms. The spell out the names of the APL primitives to get the names of the Java methods; example: the shape primitive is spelled “shape.”

There results are a caution about the imbedding approach. Their first example is a one-line APL expression that removes duplicates from a vector. The equivalent APL code has 29 lines. The equivalent code in a C-like language is two lines:

```java
function unique(generic array v)
    return reduce((v index v) = index shape v), v)
```

Examination of the authors’ example suggests that a lot of the increase from 3 to 29 lines is caused by the need to setup the Java objects. Here is a re-written version of the author’s code that is tighter (and is written in a pigdin version of Java):

```java
method unique(APL v) {
    return (v.index(v) = v.shape().index()).reduce(v)
}
```

This is two lines, better than C-like’s three, but is (in my view) much harder to read and type.

7 Extensions to APL

7.1 Brown and Others-2000: Object-Oriented APL

Robert B. Brown and his co-authors described in 2000 one approach to adding object-oriented feature to APL [Bro00b]. They conclude that “APL can be extended with OO in reasonable ways.” Their work was motivated by their conclusion that no serious effort had been made to integrate OO facilities into APL. They finished their design work in 1997 but published in 2000, so others may have pushed forward similar ideas. We ignore the details of the proposed syntax and focus on their semantics.

A class is declared in a manner that is similar to APL’s declaration of functions. One specifies the parameters to be used by the constructor and the names of the visible methods. Private variables are created by simply assigning to them in the constructor. A constructor is a method with the same name as the class. The constructor is called after the instance is materialized and simply assigns initial values. Objects are created by invoking the constructor function. Destructors are provided, but how to specify them syntactically is not covered. Class methods are named by concatenating the class name, a special symbol (an “@”) sign, and the method name. The authors spend some time discussing APLs primitive data types. The primitive types are usually given as character and number. The question is: is
an object another primitive type (they say “yes”) and does it matter that there are then lots of primitive types. If objects are primitive types, are they first class? They conclude that introducing object to APL changes “APL profoundly.”

A novel feature of their approach is that method declarations may include pre and post conditions that the run-time verifies. If one of these conditions does not hold, a domain error is signaled.

8 APL on Parallel Computers

One of our goals is to be able to move programs across diverse parallel architectures without having to change the source code. Hence a natural question is: Can APL be made to run efficiently on parallel computers?

8.1 Willhoft-1991: Most APL2 Primitives Can Be Parallelized

Roger G. Willhoft analyzed in 1991 [Wil91] how much parallelism was implicit in APL2, reaching these conclusions:

- “APL2 exhibits a high degree of parallelism.”
- “94 of the 101 primitive APL2 operations can be implemented in parallel.”
- “40-50 percent of the APL2 code in ‘real’ applications is parallel code.”

These observations suggest that if a language provides the semantics of the APL2 primitive operators the resulting applications will be 40 to 50 percent parallel. Other researchers have noted that fully-parenthesized APL expressions that are side effect free can be executed in parallel. More generally, a global data flow analysis can be used to find subexpressions that do not have mutual dependencies. Including this optimization in a compiler would increase the parallel code beyond the 40 to 50 percent implicit in APL2.

Willhoft also provides some guidelines for enhancing APL2:

- If APL2 were able to specify that functions were side-effect free, a more parallel version of the APL2 each operator would be possible.
- APL2 is not uniform in which primitives can specify the axis for operation. Thus, programmers sometimes manipulate data structures so that an axis-aware primitive can be used. Willhoft suggests that all operations accept an axis specification or that an explicit axis operator be introduced. (In APL, an axis operator would take another function and modify it so that it was axis aware.)
• APL2 lacks control-flow mechanisms, except for the goto statement and the each operators (which implements a forall construct.) Willhoft says that introducing the traditional control flow constructs (which he enumerates as looping, recursion, if-then-else, and where) would not only improve the readability of the code but also have “obvious data-flow simplification.”

Willhoft defines four types of parallelism and discusses how APL2 might exploit them.

Data parallelism is defined as “the application of a single conceptual operation to a number of data items at the same time,” where each operation “is completely independent of the rest.” Because arrays are conceptually a unified whole in APL, a lot of this type of parallelism is available. Willhoft goes on to analyze each APL2 operation and notes that APL can readily take advantage of some parallel hardware designed for general purposes, for example, the IBM 3090 Vector Facility appears to one observer as “a machine designed for executing APL.”

Algorithm parallelism is defined as “operations that can exploit the relationships of the data items to allow execution in parallel . . . There are suboperations that can be executed in parallel, but these [suboperations] must be coordinated and supervised by an overall plan.” Examples: sorts, FFTs, matrix inversions.

Data-flow parallelism is defined as “parallelism resulting from the flow of results of one operation to arguments of the next operation. Since often there are multiple arguments to a given operation, each of those arguments can be calculated in parallel.” A data-flow analysis is needed to find all of the possibilities. Explicitly notating these opportunities in the program code is tedious and error-prone for the programmer.

Task parallelism is defined as parallelism in the form of “separate tasks that are started and stopped by the application. These tasks run concurrently and may or may not communicate and synchronize with each other. All other forms of parallelism can be broken down into task parallelism.” In APL2, the shared variable facility is used to implement task parallelism.

Willhoft describes language features that are “hindrances to parallelism.”

Assignment and side effect Willhoft believes that parallelism is easier to achieve if the language has single assignment and is side-effect free.

Dynamic binding Binding names to objects at run-time is obviously more expensive than binding names at compile time.

Branching (arbitrary goto’s) APL’s reliance on the goto statement makes analyzing basic blocks more difficult.
Declarations

Declarations are necessary in APL in some instances in order for the compiler to know the element type and shape of each array.

Willhoft then classifies and discusses each APL2 primitive, concluding that 94 of 101 of them can be parallelized. We list his categories here for reference purposes.

1. Monadic scalar functions. A scalar function is a function that “operate[s] on individual elements of an array in exactly the same way that [it] is applied to the entire array.” These functions are embarrassingly parallel.

2. Dyadic scalar functions. These are also embarrassingly parallel.

3. Right scalar functions. Also embarrassingly parallel.

4. Left scalar functions. Likewise.

5. Scalar-derived functions. These are the functions derived from the each operator, which has semantics similar to forall.

6. Approximately scalar functions. These functions do not meet the definition for scalar but nonetheless can be parallelized, perhaps with some cleverness. Example: indexing is in this group and its parallel implementation would involve “several sequential steps” each of which could be parallelized.

7. Reduction functions. Guy Steele is said to have “called them primitive parallel operations.” A reduction is a special case of a scan.

8. Scan functions. N. Brenner is said to have devised a way to parallelize a scan in $\log_2(s)$ operations where $s$ is the shape of the input vector. (The exact formula is not clear in the original but the $\log_2$ term suggests some type of divide and conquer approach.) Thus “a million element vector can be scanned [and hence reduced] in 20 [parallel operations].”

9. Product functions. These include inner and outer products. Portions of the calculations can be executed in parallel.

10. Axis functions. These are functions that apply to subarrays of their arguments which can be parallelized.

11. Recursive primitive functions. These functions generate a tree-structure and hence the leaves can be evaluated in parallel.

12. Matrix inverse functions: divide, invert. These can be parallelized.


14. Rearrangement functions. Examples: drop, reshape, take, transpose. All can be parallelized.

The handful of APL2 primitive functions that cannot be parallelized are:
1. Deal: defined only on scalars.

2. Enclose, first, pick, shape: each does a single operation on an entire array.

3. Execute: calls the APL2 interpreter on a character string (the evaluation of the string itself can be parallelized).

Other opportunities to parallelize APL2 code mentioned by Willhoft are:

- Vector notation, in which a vector is specified by placing its components syntactically adjacent to other components. The components, if expressions instead of constants, can be evaluated in parallel.

- Data-flow analysis would find expressions that are not mutually dependent and hence can be parallelized.

Willhoft analyzed three workspaces to measure the actual amount of parallelism. The workspace were for these APL applications:

- Database verification. There were 5000 records that were read and verified for consistency via lookup tables and syntax checks.

- A course enrollment system that was interactive. In the measurement run, two courses were available.

- An application that displayed presentation, business, and scientific graphics. The application was the demo program that was part of GRAPHPAK, a workspace that was bundled with APL2. The code was described as written in a looping style and hence more sequential than necessary.

These applications do not seem representative of the types of applications that we target, however, we report Willhoft’s results which are not discouraging:

- The average number of elements per array function was 10 to 100.

- The percent of operations (functions) that were in Willhoft’s parallelizable group was 45 percent.

### 8.2 Bernecky-1993: APL Can Be Parallelized

In a paper presented at ACM SIGAPL APL93, Robert Bernecky (the designer of APEX, an APL compiler and for many years the head of product development at I.P. Sharp, the Canadian APL vendor) made the case that APL could be parallelized. Our remarks on Bernecky’s content [are in square brackets].

He starts with observing that “extraction of parallelism from scalar languages is possible but limited . . . since . . . any parallelism inherent in the algorithm is discarded by the limited expressiveness of the language used to describe the
algorithm.” He continues: “Most adaptations of scalar languages to parallel expression have been done from the standpoint of a particular machine design [this was in 1993, before OpenMP and MPI], and require that the application programmer explicitly embed those architectural assumptions in the application program.” [Even with OpenMP and MPI, the programmer makes explicit assumptions about the architecture. What is needed is a way to describe the algorithm and then separately describe the architecture and how to exploit it. Thus potentially two languages are needed.]

“APL [and other languages with similar semantics] has been shown to possess an immense amount of parallel expression, providing a rich blend of SIMD, MIMD, and SPMD capabilities. It does this without compromise—there is no need to embed machine characteristics within application programs.” Moreover, “APL’s richness of expression provides significant semantic content for the compiler write, which makes generation of high-performance code an easy task.”

Bernecky argues that APL and J “do much” to solve the problem of expression the algorithm in a fashion that allows parallelization. He cites three attributes of APL that make this possible: array orientation, adverbs and conjunctions, and consistent syntax and semantics.

His definition of array orientation is that all primitives are defined by their actions on entire collections (arrays) of data. As a demonstration of the power of defining primitives on entire arrays, he notes that the scalar primitives of APL (these are the subset of primitives given in Table 5) by definition (of scalar) work on each element of their operands independently and hence are inherently parallelizable.

APL2 introduced the notion of adverbs, which are simply functions that take other functions as input and modify the behavior of the other function in a uniform way. One example is the rank adverb, which “specifies the dimensions of the arguments to which a specific verb (function) is to be applied.” An example is that by using rank and reverse, one can specify the reversal of rows or the reversal of columns without writing two separate reversal functions.

Bernecky highlights the importance of adverbs: “10 verbs and 20 adverbs give the potential for specifying 200 actions.” Adverbs “are [in essence] macros which allow the user to fill in the blanks for a specific computation.” For example, in the inner product, the user specifies the combining and reducing verbs, while the control structure remains unchanged.

Consistent syntax and semantics are not actually covered in the paper. However, APL1 was known for a uniform syntax and we believe a lot of the uniformity was sacrificed in APL2 as nested arrays were added to the language. The author also states that when the relational operators are scalar and hence can operate on arrays a component at a time, some conditional and loop statements can be avoided. The example given is a J expression that adds 5 to the elements of array eo that have the value 2 mod 3. This statement in J has no loops and no if statements. The significance of this argument is that “control structure dependencies often stall pipelines, whereas data dependencies do not.” [Nonetheless, knowing the size of the dimensions of eo...
<table>
<thead>
<tr>
<th>J Notation</th>
<th>Monadic Names</th>
<th>Dyadic Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>conjugate</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>negate</td>
<td>subtract</td>
</tr>
<tr>
<td>*</td>
<td>signum</td>
<td>divide</td>
</tr>
<tr>
<td>%</td>
<td>1 divided by</td>
<td>divide</td>
</tr>
<tr>
<td>.</td>
<td>exp(y)</td>
<td>$x^y$</td>
</tr>
<tr>
<td>.</td>
<td>ln y</td>
<td>$\log_{x}y$</td>
</tr>
<tr>
<td>&lt;.</td>
<td>floor</td>
<td>minimum</td>
</tr>
<tr>
<td>&gt;.</td>
<td>ceiling</td>
<td>maximum</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td>less than</td>
</tr>
<tr>
<td>&lt;:</td>
<td></td>
<td>less or equal</td>
</tr>
<tr>
<td>=</td>
<td></td>
<td>equal</td>
</tr>
<tr>
<td>&gt;:</td>
<td></td>
<td>greater or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td>greater</td>
</tr>
<tr>
<td>+.</td>
<td></td>
<td>logical or</td>
</tr>
<tr>
<td>*.</td>
<td></td>
<td>logical and</td>
</tr>
<tr>
<td>+:</td>
<td></td>
<td>logical nand</td>
</tr>
<tr>
<td>o.</td>
<td>$\pi$ times</td>
<td>sin, cos, ...</td>
</tr>
<tr>
<td>?</td>
<td>roll</td>
<td>deal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abs value</td>
</tr>
</tbody>
</table>

Table 5: Selected APL and J Scalar Primitives
would help in selecting a loop or vectorized implementation.]

The verbs (built-in functions) of APL are defined to include such operations as transposing, rotating, and reversing arrays, where the array can be of arbitrary dimensions, size of dimensions, and elements. The machine architecture can be used to find the best algorithm. The author’s example is a search operation in APL, where a binary search might be the best algorithm on a Cray X-MP and a straight check-everything is fastest on a Connection Machine.

Topics Bernecky mentions as important to low-level optimization are: cache management, data distribution (he focuses on FORTRAN’s column-order versus C’s row order, but the issue also includes placement within memory), code scheduling, and loop unrolling.

The author cautions against allowing pointers, especially where the referent’s type is not known. The problem isn’t cache misses and the like, but that with such pointers, determining if a variable is aliased is difficult or impossible, and that difficulty inhibits code optimization.

Bernecky is not happy with FORTRAN 90’s inclusion of pragmas for the placement of arrays near to certain processors. He dislikes the mixing up of algorithm and implementation and wonders if the pragmas are necessary because of “Fortran’s semantic poverty.”

He gives an extended example of the flexibility that APL has in allocating an array. To avoid cache interference (assuming multiple processors are to be used), the compiler needs to determine how to store each array. “It might, for example, append extra columns to an array . . . It might broadcast multiple copies of array segments to different processors.”

One technique the author proposes is to allow the program to specify assertions regarding the array’s properties, for example, the array might be symmetric or sparse or upper triangle. The compiler can potentially use this information to select fast algorithms.

Bernecky offers a catalog of APL-related parallelization opportunities that would apply to any language with similar semantics:

- Scalar functions. These are functions that apply element-by-element to arrays. Many are predefined (see Table 5). (APL calls these rank-0 verbs.) The execution of these verbs can always employ fine-grained parallelism because the result is built up element by element.

- Non-scalar functions. In APL, these are primarily used for selection, structuring, and search on arrays. Bernecky claims that many of these functions have parallelizable implementations. For example, he says that there are opportunities to sort in parallel.

- “Operation of a rank-k verb upon arrays of higher rank than k is defined as independent application of the verb to each rank-k subarray of the argument.” Thus, parallelizable, even without knowing what a rank-k verb is. The rank of a verb (which is a function) is the number of axes in the arrays to which the verb “naturally” applies. Rank can range from 0 to infinity. Bernecky’s example may help to explain: consider an array of
size 2 x 3 x 5 x 4 and suppose we want the inverse. “Matrix inverse applied independently to each of the six 2 by 3 cells of shape 5 x 4 to produce a result of shape 2 x 3 x 4 x 5. That is, each inverse produces a result of shape 4 x 5, and the six (2 x 3) of them are laminated into the 2 by 3 frame.”

• APL also supports extension: “when one argument to a dyadic verb is the same rank or less than the defined rank of the verb, that argument is extended by reusing it as many times as necessary.” (All verbs have a defined rank. There is an adverb called “rank” that allows this rank to be redefined in a given application of the function.) Extensions provide parallel opportunities.

• APLs adverbs shorten programs. Some of them (perhaps all) allow for parallel operations. The example given is the APL scan adverb, which can be used to implement recurrence relations. Another example is the rank adverb.

• APL’s expressions, when fully parenthesized, can be run on as many parallel execution threads as there are innermost expressions. This is called parenthetical parallelism.

• Chaining (also called loop jamming) allows “merging of a sequence of operations on primitive functions [built-in functions] on arrays into an interleaved execution on subsets of the arrays . . . Loop jamming has the potential for significant performance [improvement] in an interpretive environment [and perhaps also in a compiled environment], because it reduced the load/store traffic associated with array-valued temps, as well as eliminating the storage management overhead associated with each jammed primitive.” [A compiler might optimize away the array temps.]

• APL’s fork and hook syntactic forms join a sequence of operations on two arrays into one syntactic unit. The sequence can be parallelized. Example: A (f g h) B is a fork that is short-hand for (A f B) g (A h B).

• Composition in APL is a way to get the Unix shell’s pipeline functionality. In an APL pipe, which is in the pipeline is an entire array. This allows for MIMD parallelism because “each cell can be computed independently in any of the composed verbs.”

• Gerunds, a feature of J, allow parallel execution, because a gerund is simply an array of functions that is then applied to actual numeric data. This allows for MIMD computation.

8.3 Bernecky-1997: APEX, An APL Compiler

Robert Bernecky was manager of APL development at I.P. Sharp Associates Limited, a major APL vendor, in the 1970s [Ber93]. Subsequently, drawing in
part on his experience there, he wrote a master of science thesis on APEX, an APL compiler that had acceptable performance compared to APL interpreters (this is fairly easy) and compiled scalar languages (like FORTRAN) [Ber97]. His APEX compiler is available today, though its still in experimental form and doesn’t appear to have gained significant commercial traction or use. Bernecky summarizes: “The excellent run-time performance of APEX-generated code [the APL compiler] arises from attention to all aspects of program execution:

- run-time syntax analysis is eliminated,
- setups costs are reduced,
- algebraic identities and phrase recognition detect special cases,
- some matrix products exploit a generalization of sparse-matrix algebra, and
- loop fusions and [array] copy optimizations eliminate many array-valued temporaries.”

8.3.1 Approach and Results

Compared to interpreted APL, Bernecky summarizes by saying (page 54) “For highly interactive [APL] applications, APEX can produced code that executes several hundred times faster than interpreted APL, where as small, straight-line APL applications may execute at similar speeds in both environments.” Thus, APEX compilation results in executions times that are never worse and often much better.

Compared to compiled FORTRAN programs, Bernecky summarizes by saying (page 83) “Compiled APL kernels [meaning a sample of benchmark codes] and applications [entire programs], particularly if assisted by the use of Extended APL language features [features that Bernecky introduced to aid the performance of compiled code], can approach and sometimes exceed the performance of FORTRAN, but that array copying . . . still remains a major hurdle for some matrix-based applications.” The last point will be explained below but first the next level of detail. Comparing G77 (the GNU FORTRAN 77 compiler) with APEX-compiled APL on an Intel 486, the APEX code was slightly faster. Comparing F77(Sun’s FORTRAN 77 compiler) with APEX on Sun system (details not given), the APEX code was much slower. Perhaps Sun has invested to make sure that FORTRAN is fast on their systems and the gnu project has not carried out the same investment for the 486 processor, but this is just speculation.

On the array-copying point, which seems to have hurt the APEX compiler on the Sun systems, Bernecky traces the problem to his compiler strategy. Rather than compile APL to assembler, he compiles APL to SISAL (a functional language). SISAL emits C code which is subsequently compiled. The array-copying problem is that SISAL does not implement matrices and higher
order arrays as blocks of storage, but rather as vectors of vectors. The result is that some APL primitives including take and drop, both used to restructure arrays, are implemented inefficiently in the SISAL-emitted code and not optimized sufficiently by the C compiler. If blocks of storage were available in SISAL, Bernecky states (page 70) that restructuring operations on arrays could be implemented through array coordinate remapping and not through array copying, as the use of SISAL forces. This analysis seems plausible and hence gives hope that compiled APL could perform roughly as well as compiled FORTRAN. (Later, according to his web site, Bernecky reworked APEX to generate Single Assignment C [Ber09b]).

Note that even if the run-time of compiled APL is close to that of FORTRAN, the presumed faster design and coding time from using a high-level language like APL could make the overall business case for compiled APL favorable. Bernecky attempted benchmarks to assess the extent to which APEX’s parallelized code achieved linear speedup with the number of processors available, but was not able to obtain anything but heavily loaded systems to test on. Hence, his measurements may not indicate what is actually possible. For some processors and workloads, 8 processors achieved about 80 percent of a linear speedup; for others, the speedup for 8 processors was closer to 50 percent. He was not able to do side-by-side comparisons of say a FORTRAN code compared to an APL code, so how much of the fall-off in linear performance was due to the algorithm and how much was due to the implementation isn’t known.

The APEX compiler required several minutes to compile an application. The reason for the slow performance was that it was written in APL and executed under an interpreter running on Windows and hence on a personal computer. A compiled version of the compiler would have better performance.

8.3.2 Achieving “Portable Parallelism” with APL

Bernecky also studies APEX’s ability to parallelism programs. He has high hopes that APL compiled APL will provide a means of achieving portability and parallelized code, based on the conclusion that compiling reduces about 50 percent of the overhead that interpretation introduces and that (page 95) “the abstract nature of APL, whereby a user specified what is to be done in a computation, without having to express, or even know, how the computation is actually performed” is the key to “portable parallelism.” He notes that this property of APL “permits a programmer who knows nothing about parallel computing to write parallel programs” (page 96).

He goes on to state three reasons “that hand-crafter parallelism may not, in practice, deliver maximum parallel performance:”

- “The ideal expression of parallelism for one computer architecture may be highly inappropriate for another.” Thus, a programmer may need to maintain a version of an application for every architecture that it is run on.
• “Hand-crafter parallelism confounds expression of the application with expression of the parallel systems architecture.” Thus the code talks both about the algorithm and the architecture.

• Development schedules may prohibit manual tuning for parallelism as might the available skill set of the programmers.

8.3.3 Why APL Interpreters are Slow

APL is said by some to be slow. Here are the reasons given by Bernecky:

• “Each name in an APL application can theoretically change meaning from one instance to another.”

• The first three of four stages in executing an APL statement involve run-time checks. These stages are: (1) syntax analysis, (2) conformability checking, (3) memory management, and (4) primitive-specific operations. According to Bernecky’s analysis, the first three stages (called collective setup costs) in one well-written APL application running in production, fully half of the CPU time was consumed by setup costs. In another application, 2/3’s of the CPU time was setup costs.

APEX accomplished array element and shape type checking by doing a global flow analysis and deducing the rank and element content of most arrays and requiring the programmer to insert declarations (as part of comments) for the arrays whose rank and element type cannot be deduced. Bernecky calls his process array morphology.

To reduce memory management (stage 3), APEX employs traditional compiler techniques including loop fusion, copy avoidance through reference counting, and special treatment of scalars. Loop fusion means combining two loops so that only one index is used. Copy avoidance arises in array languages as assignment to an element of an existing array must generate a copy of the existing array, if there are any other referents to it. A reference count can be kept to keep track of the number of referents. APEX avoids reference counting through a global flow analysis.

To reduce the time for the execute phase, interpreters can determine and recognize special cases; example (Bernecky): APL’s `indexof` function “is frequently used to search for a single value, rather than a whole array of values . . . In such a case, a sophisticated search algorithm usually does not pay off; a simple linear search is most effective.” Another technique used by APL interpreters is idiom recognition. For example: there is an APL expression that is often used to eliminate duplicate items from a vector. A third technique used by interpreters is to map array coordinates rather than actually restructure the array. These three optimizations can be more easily done by a compiler rather than an interpreter, as the compiler can afford to invest more time. One execution speedup that only a compiler is likely to do is to inline code.
• Naive APL interpreters create temporary array-valued intermediate results that are discarded by the time the entire statement is executed. About half of the arrays processed in a typical APL program have zero elements or one (page 14), an interpreter can often afford to detect these special cases and speed them up.

8.3.4 APL Language Features That Complicate Compilation

Bernecky catalogues language features of APL that increase the level of difficulty of compiling it.

• Array type and rank determination. These must be deduced by an APL compiler.

• Defined function overloading, which in APL means, that a given function might be called one time with a vector as argument and another time with a matrix as argument.

• Variable type and rank conflict

• Dynamic scoping of names, which makes finding certain classes of bugs in APL extremely difficult.

• Semi-globals are variables that APL programs use in large part because a function may have only two explicit parameters. The semi-globals are implicit parameters.

• Unrestricted goto, which is the primary flow of control mechanism in APL.

• Side effects (defined as using variable to store state information), an artifact of programming style in APL that leverages the goto statement, which can be conditional and in-effect a computed goto.

• Dynamic nameclass changes. In APL a variable might be bound to an array in one interpretation of a statement and a function in another.

8.3.5 Speeding Up by Recognizing Arrays in Special Form

APEX takes advantages of special cases for arrays when processing some APL operations.

For APEX, the special cases are detected via a global flow analysis. APEX’s mechanism for defining the special cases is called array predicate, which are one or more properties associated with an array.

The array predicates APEX detects are:

• PV: array is a permutation vector (the elements 0, 1, . . . , N have been permuted)

• PVSubset: array is a subset of a permutation vector
• NoDups: each element is unique
• All2: each element is the integer 2 (these vectors are used for converting numbers to and from binary representation)
• SortedUp: elements are in increasing order
• SortedDown: . . . decreasing order
• KnowValue: all element values are known at compile time
• NonNeg: each element is non-negative
• Integer: each element is an integer

8.4 Ching and Ju-1991: APL on RP3

W.M. Ching and D. Ju published in 1991 a description of their approach and results on parallelizing APL on RP3, a 64-way MIMD system built by IBM for research purposes [CJ91].

RP3: A 64-way MIMD shared-memory machine built by IBM at their Thomas J. Watson Research Center. It was never released commercially. The machine contains 64 process-memory elements (PMEs) connected to a network. Each PME has a processor, eight MB of memory, and a 32 KB cache. All memory can be directly accessed by all processors though access to memory outside of the PME requires use of the network. The network was characterized as “very fast . . . and usually not a bottleneck.”

APL Version Used: A conventional APL program with a few restrictions on the data actually used. The restrictions were in the scope of the language: exclusion of APL’s execute function (which invokes the interpreter on a string), no support for system functions (example: input-output functions), support for only a few APL system variables, support for only a few APL branch expression forms. And there were restrictions on the actual array assigned to a variable: after the initial assignment, the array must remain either numeric or character and its rank cannot change. The rank was limited to no more than seven dimensions. Of note, no declarations are required. Some features of APL2 were not included, but these were judged to be straight-forward extensions of the presented work. These features were: complex numbers, the each operator, and grouped parameters. In contrast, supporting the nested array feature of APL2 was judged to “require far more extensive work” (page 769).

Why APL: “The succinctness of a well-written APL program implies that there are no incidental or artificial constraints on the order of execution. These constraints may well occur in FORTRAN programs . . . . It is quite easy [in APL programs] to determine which parts may be executed in parallel.” (page 768)
Compilation Approach: The researchers already had an APL compiler which generated System/370 assembler code. It was modified to generate C code. The C code was then compiled for the RP3 system. RP3 ran the Mach operating system and the compiled code used Mach’s threads. Their compiler was incomplete in that it did not implement all of the APL primitives.

The front end of their compiler parses the program and does “interval-based dataflow analysis” in order to deduce a “type-shape” of the program’s variables. Some variables have known dimensions at compile time, other unknown, with the exclusion of the parameters to the main program. Except for the parameters of the main program, variables are classified by the front end as being of type Boolean, character, integer, or float and with shape scalar, array with known dimensions, vector of unknown length, or array of unknown dimensions. The front end analyses basic blocks to determine data dependencies. The dependencies are used to generate “send/wait synchronization flags at parse-tree nodes”. These flags are used by the back end in exploiting the inherent parallelism in the program.

Key source of parallelism is the execution of the primitive operations. The version of the Mach operating system used was modified to enable low-overhead kernel threads that are bound to processors (and hence provide a way to leverage memory and cache-consistency).

Parallelism is achieved in the execution of the primitives. If a run-time check indicates that a parallel execution is worth the overhead, the runtime assigns previously allocated threads to components of the array. The array is then divided into pieces of equal size, except perhaps for the last piece. Execution is then forked across the pieces and the former main thread is assigned the last piece. When all the forked threads have completed, they are joined and execution is once again sequential until another primitive is executed. The threads are synchronized through distributed locks in an effort to avoid memory contention. The join operation was implemented via a single variable. Each thread increments the variable when it is finished and the main thread checks the variable to determine if the worker threads have finished.

Speedup Constraint: Amdahl’s Law provides a constraint. Example: if 10 percent of a program must run sequentially, then parallelizing the remainder with an infinite number of processors so that the elapsed time goes to zero still requires 10 percent of the original execution time, so that the maximum speed up a factor is 10.

Speedup Results: For three example programs, the best parallel speedup obtained was a factor of 12 for the entire program. However, considering the execution only of the primitives (the only portion parallelized), a
speed up of as high as 32 was observed. Recall that the RP3 system has 64 processors.

**Beliefs About Even More Speedup:** The authors hypothesize that more speedup would be obtained through caching and a better layout of the data in memory and use of parallelism beyond the basic APL primitives and via primitive fusing.

- Caching was a problem on the RP3 because of the way floating point was implemented which forced many array element accessed to miss the cache.
- Better memory layout would involve storing the arrays portions in the local memory of the PME’s. In the experiments described, the arrays were stored in shared memory and hence memory access required going through the network. Though the network was described as fast, it nevertheless had congestion.
- The experiment parallelized only the APL primitives. However, their compiler was able to determine when two of the parse subtrees could be executed in parallel and this potential parallelism was not exploited.
- Primitive fusing would analyze a consecutive set of primitives. Rather than parallelize each primitive on a stand-alone basis, the primitives could be re-written on potions of the arrays and the rewritten sequence executed in its entirety.

Some notable conclusions:

- “The prospect of the extra work in partitioning an application for parallel execution and orchestrating the communication among the segments is daunting enough to dampen the ordinary user’s enthusiasm for parallelism . . . . For the ordinary user, parallelism is interesting only if it is transparent.” (page 767, emphasis is the original)

- Define an APL-style program as one in which a major portion of the code is expressed in high-level array operations. The authors observe: “APL-style programs for uniprocessors do not tend to run significantly faster when compiled and executed than when interpreted.” (page 776) They can know this because they have an APL compiler.

- “A good APL-style program that enjoys a moderate sequential speedup [when compiled and executed] on a uniprocessor exhibits better parallel speedup than a sequential-style program [when parallelized on a multiprocessor].” (page 776)

- “The programming style APL programmers developed to avoid inefficiencies in the interpreter is precisely what makes their programs naturally suited for parallel execution.” (page 776)
• The effect of interpreting the sequential portions instead of compiling them can be significant on speedup. Example (take from page 776): a program spends 10 percent of its time interpreting, 10 percent executing sequential code, and 80 percent executing parallelizable code. When interpreted, the maximum speedup is a factor of five. If the interpreted portion goes to zero, the maximum speedup is a factor of nine. Thus, in the example, the speedup is doubled through compilation.

8.5 Greenlaw and Snyder-1991: APL on SIMD Machines

Raymond Greenlaw and Lawrence Snyder in a report published in 1991 [GS91] leveraged Saal’s static analysis of APL program [SW75] to estimate the speedup for APL programs. Saal demonstrated that “nearly 95 percent of the operators appearing in APL programs are either static primitive, reduction, or indexing.” Greenlaw and Snyder show that if these operators occur at least 95 percent of the time in the execution of typical APL programs, then “substantial speedups” are possible, as demonstrated by both an analysis and empirical measurement. They state that their conclusions apply to SIMD distributed memory machines.

APL is not parallel strictly speaking, because the semantics of program execution are based on sequential execution. What is parallel is that the many built-in operators (the APL primitives) are inherently parallel. Another parallel-like feature of APL is that the programming style usually adopted is similar to that of functional languages. [In fact, other researchers would say that APL programs are not typically functional, because the limitation of at most two parameters per function causes programmers to use side effects in the form of assignments to global variables frequently.]

APL is SIMD friendly in that “the semantics of APL are very compatible with the computing capabilities of SIMD parallel computers, especially those with nonshared memory.”

A simple parallel compilation strategy for SIMD nonshared memory systems is to distribute each array among the processors. Most of the values resulting from an operation are retained in the processor that holds the portion of an array. When this is not true, communication among the processors is required.

The central question in this strategy is: “How much performance improvement can one expect from this simple implementation of APL?”

The approach to answering the question is to focus only on the three classes of operators that make up 95 percent of typical APL programs (the scalar primitives, the reduction operator, and the subscript operator). Within this focus, Greenlaw and Snyder perform both an analysis and an empirical study.
Machine architecture assumptions used by the authors may not hold for more modern machines (their work was published in 1991), so it is described here. In addition to the computational processors, there is a control processor that broadcasts to each computational processor a single instruction stream. Each processor (including the control processor) has local memory and keeps a local APL symbol table in its memory. APL arrays are distributed across the memories of the computation processors such that an array $A$ of size $n$ has element $A[x]$ stored in processor number $1 + (x \mod P)$ where $P$ is the number of processors. Some stored arrays in the last processors in the numbering scheme are extended so that every stored portion is the same size (so that a single instruction stream can be used). Further, each processor can access the memory of any other processor in a single step, a feature that did not hold for all SIMD systems of the 1991 era (nor today, what with NUMA machines).

They analyze speedups using Amdahl’s law and a specific set of algorithms suggested by their interconnect and alternatives then in use. The speedup in going from 1 to 1024 processors is about 110 times in one case (thus about 10 percent of an embarrassingly parallel result) and about 660 in another case (about 66 percent of an embarrassingly parallel result). These results are a kind of upper bound on speedup achievable with the “$x \mod P$” data distribution. [Note: other researchers have distributed the data in different ways.]

Greenlaw and Snyder then simulated the compiled APL programs and determined that for large arrays, “the speedups achieved are roughly 65 percent of the maximum possible.”

The effect of not parallelizing the sequential code in APL programs can be seen from the following analysis. Assuming large arrays and use of 9, 81, and 256 processors. Then the speedups were:

- for 9 processors: 91 percent of maximum theoretical
- for 81 processors: 73 percent
- for 256 processors: 48 percent

8.6 Ching and Katz-1994: APL on MIMD Machines

Wai-Mee Ching and Alex Katz in 1994 described how to compile APL onto a MIMD machine programmed with a SPMD approach [CK94].

The machine was an IBM SP1, a commercially available multiprocessor RISC machine running IBM’s version of Unix. The machine has distributed memory and hence a network. This particular network provides any-to-any connectivity such that “the time to send a byte of message from one processor to another is almost identical for any pair of processors.” The “flat” nature of the interconnect network simplified the compiler.
The language accepted by the compiler was APL with added declarations for the input arrays. The element type and shape of each other array was deduced statically through a global flow analysis. Only flat arrays were supported. (Ching notes that flat-array APL can be “compiled with efficiency comparable to that of FORTRAN programs, citing his own work [CNS89b]. Others have claimed that the nested arrays of APL2 greatly increase the complexity of compiling APL.)

The compiler was a modified version of Ching’s previously-created APL compiler for a shared memory MIMD machine [JWC94]. Ching describes the modifications as not being extensive, which suggests that a compiler for an array language could target both shared MIMD and distributed MIMD machines. The starting-point compiler had about 12,000 lines of APL code. The compiler produced C code which is itself compiled by the system standard compiler. The generated C code used an SP1-specific library that leveraged a custom-designed chip (“eui”) that implemented primitives that appear similar to the MPI primitives. Ching’s compiler used the MIMD machine in a single-program multiple-data (SPMD) programming model.

The key compile-time decision is to “decide which program variables are to be replicated and which are to be partitioned.” The short approximate answer is: partition large variables, which are defined as

- any variable known to be larger than a threshold size
- any variable whose size is not known statically (these variables are typically input variables)
- any variable derived from a large variable
- any temporary result derived from certain APL primitives that always generate large values unless their input is constant (example: the iota operator).

This last criteria introduces complications, so Ching defines a more refined algorithm to decide which variables are replicated in each processor and which are distributed (partitioned) across processors. He describes the algorithm in two steps, which are collapsed in this presentation. Every variable is divided into one of four classes (class names have been edited for clarity):

- “Partitioned: each processor holds a section of the array, divided along some axis of the array.”
- “Replicated: each processor hold a copy of the array.”
- Replicated Scalar: the array is replicated and is a scalar. These differ from partitioned arrays in that an APL binary primitive operators when applied to them may yield a Partitioned or Replicated result depending on whether the other operand is Partitioned or Replicated.
• Partial: Only one processor holds the array. One combination of APL primitive and specific data types generates this situation: a Partitioned array is indexed (subscripted) along the partitioned axis (which is often the first axis of the array) and the index is a Replicated Scalar.

To determine which arrays are Partitioned, the following definition is applied:

• It is a large input to the top-level function. Ching’s compiler determined this by a declaration.

• It is the result of an expression $W \ op \ V$, where $op$ is a scalar function and either $W$ or $V$ is Partitioned.

• It is the result of an expression $op \ V$, where $op$ is any primitive except $\rho$ (shape) and $V$ is Partitioned.

• It is the result of an expression $W \ op \ V$, where $V$ is Partitioned and $op$ is one of six specific primitive operators.

• It is the result of an expression $W \ op \ V$, where $op$ is an outer product of the form $\circ f$ for some scalar function $f$ and $W$ is Partitioned

• It is on the left side of an assignment statement and the right side is an expression $W[V]$ where $W$ or $V$ is Partitioned.

An array is Replicated if it is not Partitioned.

The next compile-time decision is how to distribute the large variables. Ching’s compiler always distributes the large variables by partitioning them on their first axis unless they fall into one a few special categories, where another axis is used. First-axis partitioning results in the data in the partitions holding contiguous elements in an array, which is good for code generation and speed. For many distributions, the compiler adds dummy elements to the end of the last array partition, generates code to operate on these elements, and then generates code to exclude (or neutralize) these results from the rest of the computation. The added elements are necessary because the exact same program is executed on each processor and the arrays cannot always be evenly partitioned across processors.

Ching notes that the size of the arrays determines in part whether they are best replicated or partitioned. We add: Consider the implications for a code that is run on data sets of multiple sizes. Sometimes a certain array should be replicated, sometimes partitioned. The choice depends on the input. Thus a human who decorates the code with partitioned-replication decisions faces the work of manually adjusting these decisions. To get all these decisions right requires an algorithm that is equivalent to what a compiler or its run time can do. Thus a good compiler should perform at least as well as a good human and hence better on average.

Ching’s code does achieve speedup on five small test cases, which were not production programs, but small sample codes. We report only the results for the largest input array for 32 processors. The maximum speedup assuming the programs were embarrassingly parallel would be 32 times. The results were
Case 1: 19 times
Case 2: 27 times
Case 3: 31 times
Case 4: 21 times
Case 5: 14 times
This results are certainly not discouraging.

8.7 Ju and others-1994: Loop Fusion of Array Language Primitives

Dz-Ching Ju and his coauthors presented in 1994 [JWC94] a “classification scheme for array languages primitives that quantifies the variation in parallelism and data locality that results from the fusion of any two primitives.” Our interest is in the list of primitives and the optimization. According to the Dragon Book’s Chapter 11 [ALSU07], fusion is a source code transformation that combines loops that have their own indices into one loop that uses the same index. The statements in the former multiple loops are hence fused into statements in one loop. The example given in

```c
/* before */
for (i=1; i <= n; i++)
  Y[i] = Z[i];
for (j=1; j <= n; j++)
  X[j] = Y[j];

/* after */
for (p=1; p <= n; p++)
{
  Y[p] = Z[p];
  X[p] = Y[p];
}
```

Ju and his collaborators begin by endorsing the concept behind our project: “Array primitives can efficiently support concurrent execution and improve the portability of programs by explicitly exposing the data parallelism of each operation. The higher-level semantic information provided by array primitives can be used to optimize the performance of each primitive and to combine primitives.”

The main result (besides the catalog of primitives) is to show:

- Fusions sometimes improve execution by reducing space usage, reducing synchronizations, and increasing locality of reference.
- Fusions sometimes reduce the amount of parallelism available by grouping unfavorably combinations of array elements.
A heuristic algorithm (described in the paper) can effectively determine when fusion is beneficial.

8.7.1 Array Primitives

After reviewing APL and FORTRAN 90, the paper classifies the array primitives in APL and FORTRAN 90 into 10 broad categories. We reproduce his list here. In the list, \( L \) refers to an array that is the left operand of some primitive, \( R \) to the right operand, and \( d \) to an axis in the array.

1. Scalar operations
   - Monadic, all taking one operand \( R \): abs, ceiling, exp, factor, floor, loge, log10, not, negative, pitimes, reciprocal, sqrt, trigonometric functions, hyperbolic functions
   - Dyadic arithmetic, all taking one left operand \( L \) and one right operand \( R \): add, assign, binom, div, and, eor (xor), max, min, mul, or, power, residue, sub
   - Dyadic boolean: equal, greater, less, not equal, not greater, not less

2. Reduction, of the form \( \text{reduction}(f, R, d) \), where \( f \) is any dyadic scalar operation, and \( d \) specifies the working axis

3. Scan, of the form \( \text{scan}(f, R, d) \)

4. Outer operations, of the form \( \text{outer}(f, L, R) \), which applies dyadic scalar operation \( f \) between all combinations of pairs of elements from \( L \) and \( R \)

5. Inner operations, of the form \( \text{inner}(f, g, L, R) \), which combine subarrays along the last axis of \( L \) with subarrays along the first axis of \( R \) by executing an outer operation using \( g \), and then executing a reduction using \( f \) of that result

6. Movement operations
   - Monadic, taking one parameter \( R \): transpose
   - Monadic, taking \( R \) and \( d \) as parameters: reverse (along the \( d \)-th axis)
   - Dyadic, taking two parameters \( L \) and \( R \): subscript(\( L, R \)), in which the elements of \( R \) are the subscripts to select elements of \( L \)
   - Dyadic, taking \( L \) and \( R \) and \( d \): rotate (\( L \) positions on \( R \) using the \( d \)-th axis), shift, spread (return \( L \) copies of \( R \) along the \( d \)-th axis), catenate (along the \( d \)-th axis), pack (compress the \( d \)-th axis of \( R \) using Boolean vector \( L \)), unpack (expand the \( d \)-th axis of \( R \) using Boolean vector \( L \)), drop (remove subarrays), take (select subarrays)

7. Broadmerge operations
• Monadic: matrix inv
• Dyadic: matrix divide

8. Shape operations
• Monadic: ravel, shape
• Dyadic: reshape

9. Sorting: (all monadic) grade down, grade up (the grade operators run a permutation vector that can be used to access the original array in sorted order), sort [Note: Ju’s grading operators could be extended to specify the axis used]

10. Searching: (all dyadic) index (yield the first occurrence in R of elements in L), membership (return 1 if and only if the corresponding element of L can be found in R)

8.7.2 Optimization Results
We describe the architecture that Ju works with (which appears to be an abstraction of the RP3 machine IBM had in its research lab):

• A shared-memory multiprocessor
• Each processor with its own control unit, cache, and local memory.
• Processors and global memory are connected with a network.
• In execution, a single program is run on each processor; hence the model is SPMD (single-program, multiple-data)

The main result is an algorithm that given this type of architecture, determines whether two array operations should be fused.

9 APL’s Market success
9.1 Wikipedia on APL
Wikipedia’s entry for APL claims that instead “APL was unique in the speed with which it could perform complex matrix operations” [Wik09a]. According to the article, the speed advantage came from:

• Use of highly-tuned linear algebra library routines,
• Very low interpreted overhead, because interpretation was per-array, not per-element,
• Response time favorable to early optimizing compilers
• Microcode assist (for example, on some IBM mainframes)
This article says that the optimization techniques envisioned by Abrams [Abr70] in his “An APL Machine” were not widely used. Regarding compilation, the article notes that the nested array feature of APL2 (a feature not in APL1) makes compilation “increasingly difficult.” One technique of possible interest was used by STSC for an APL compiler in the mid 1980s. This compiler produced machine code “which would execute only in the interpreter environment.” This system then went back and forth between interpretation and compilation. For heavily iterative code, “dramatic speedups did occur.”

In Microsoft’s .Net environment, VisualAPL is available. The language compiles and has access to all the .Net run-time CLR features (including data created by other programs). “Substantial performance speed-ups over legacy APL have been reported, especially when (optional) strong typing of functions is used.”

Part of the reason’s for APLs diminished success in the marketplace is its character set. It started out and continues to have many unique characters, some of which are Greek letters (like the Greek iota) and some of which are unique symbols, original chosen by Iverson to provide a mnemomic aid (example: the symbol for the operator \texttt{sort up} is an upward facing triangle with a vertical line through it). Unicode contains all of the APL characters. Here are the difficulties induced by the unique character set:

- The initial barrier to learning. This was a small issue in my view, when considered in isolation, but it led to some of the issues below.

- The decision to implement the character set by replacing the lower case letters on a standard keyboard with the special APL symbols. Thus the text of an APL program is in a mixture of upper case letters and special symbols. The use of upper case letters goes against the trend, which favors lower case letters in many languages of recent design. So APL code looks old fashioned (because of the upper case letters).

- Initially, some APL characters were entered by typing a symbol, backspacing, and then overtyping another symbol. These symbols were both harder to enter than other symbols and harder to edit.

- Getting a keyboard with the APL symbols already on it requires a special order. If one uses a laptop, then one can put stickers on the keys or just memorize where the special characters are. Both are barriers to casual usage and discourage trial.

The Wikipedia article claims that APL is still “popular in financial and insurance applications, in simulations, and in mathematical applications.” The article notes that APL usage “has steadily declined since the 1980s” for these reasons:

- Lack of support on personal computers (at least initially when the PCs came out)
• Competition from other matrix-aware tools including spreadsheets

• Competition from MATLAB, GNU Octave, and Scilab. “These scientific array-oriented platforms provide an interactive computing experience similar to APL, but more resemble conventional programming languages such as Fortran, and use standard ASCII.”

• Competition from APL’s children, including J [which enhances the semantics of APL2 and was and is free, though closed source]

9.2 Wikipedia’s Criticism of APL

We summarize Wikipedia’s criticism of APL [Wik08], with a focus on why APL has not been a huge market successes in spite of a promising beginning. The article starts with two basic observations:

• “APL . . . has been widely used since the mid 1960s on mainframe computers and has itself evolved in step with computers and the computing market.”

• “APL is not widely used.”

So, why didn’t it succeed? There is a laundry list of reasons given (and then we add our own).

The article states:

• “Acceptance or reject of APL is sometimes highly emotionally charged.” On the one hand, its interactive implementation and brevity allow very high productivity for those who are experts. We add that learning APL2 is difficult and its terse syntax, rich semantics, and unusual character set intimidate novices. (We would argue that learning APL1 is not so difficult, though one is still left with the terse syntax and rich semantics. We believe APL1 was easier to learn because its data are flat arrays, not nested arrays as in APL2, and its syntax is much more uniform: APL2 does not use strictly right-to-left parsing with an exception only for brackets and parentheses as does APL1.)

• “Even expert programmers find APL programs difficult to maintain and debug.” The terseness cuts both ways. The standard joke is: “APL is a write-only language.” (one source of many: Ben Goldberg, NYU Courant Institute Lecture, January 2009.)

• The APL character set require focus to learn and special entry methods to type. Thus the elegance of the notation is offset by practical difficulties. The article notes that the terseness of the syntax contributed to APLs initial success, as the implementations were mostly time-shared on slow computers, so that a compact representation helped performance.
- APL lacks a standard library and instead has lots of idioms to perform operations such as `remove duplicates` from a vector. These idioms are represented by sequences of APL symbols. A large part of becoming productive as an APL programmer is to learn to write and read the idioms. Note that the idioms substitute for a standard library (more on this later). The article says the best known collection of idioms is called the “FinnAPL Idiom Library” (it originated in Finland) and a web search reveals that one accounting lists 738 idioms. In my view, it's not the idioms themselves that were the issue in maintain APL in the market, but rather that they were not encapsulated in textual names (such as `remove_duplicate`). Had they been, new APL programmers would have had a lower initial learning barrier.

- The original APL editing system was built into APL itself and was primitive. In particular, changing characters was difficult. Hence programmers tend to insert characters or to append them to a line, increasing code density.

- APL uses dynamic scoping, which makes it hard to figure out the binding of a variable.

- APL from IBM typically had a flat name space, which makes sharing of workspaces challenging. Other vendors had namespaces but the syntax and semantics were not always uniform, so that workspaces could not easily be shared among users. [This decision inhibited the build-up of shared libraries written in APL.]

- Iverson's notation was developing in the 1950's before modern control structures were systematized and determined to be far superior to goto statements. Yet IBM never included control structures in its APLs. Competing vendors did, but without converging to a uniform syntax. [This decision by IBM also inhibited the build-up of shared libraries.]

- Function valence was maximized at two, forcing contortions on the part of programmers and leading to unnatural expression of the underlying algorithm. Vendors apart from IBM have extended APL to allow more parameters to functions, but once again there is not an effective standard.

- Function arguments are `in` only, but functions are not pure (as they may have not only I/O side effects but also can write to any dynamically scoped variable). Thus functions are not pure by default (as they are in Ada) nor are procedures available in IBM’s implementation (though Dyalog APL allows a function to assign to its parameters).

- APL vendors reacted to a shrinking market by extending their offers in ways that would discourage their customers from migrating. Since very few new customers were available, this action, which was sound for each
vendor, generated further fragmentation and hence discouraged people from adopting APL.

- Early APL implementations were on time-sharing systems. The initial reputation for slow performance is said (by the article) to have come from comparisons with batch executions of FORTRAN and COBOL programs. Early PC implementations of APL were even slower than the then mainframe versions. Thus the first generation of programmers who learned on PCs didn't learn APL. (Once the 80386 chip shipped, APL performance on the PC was better than on the mainframe.)

We would add these considerations to the article:

- Lack of a standard library. IBM could have published a standard APL library, building it around say the FinnAPL idioms, but did not. Hence users were own their own and since different APLs had slightly different definitions of the built-in functions and operators, not all idioms worked in all APL implementations.

- Nested arrays were introduced in part as a way to have something like a record structure (though without field names) and in part to work around the restriction that a function could have at most two arguments. But nested arrays suffer from the lack of a mathematically grounded theory and hence their semantics are somewhat arbitrary. Moreover, their introduction came at the price of greatly increased syntactic complexity.

- Poor performance, though this criticism may have been misplaced. Early APL implementations were on time-sharing systems which were often overloaded with too many users. Hence interactive performance was slow. Others (see elsewhere in this document) have argued that APL is inherently compilable and parallelizable and thus is not intrinsically slow.

- APL's bundled implementation—in which the language implementation also provided many facilities that came to be part of the operation system—also slowed improvements, because whatever investments were needed to, say, improve the file system or even text editors, had to be made at both the OS level and the APL level. Indeed investments in the OSes text editors did not directly benefit APL users, for example.

9.3 Brown-2000: What’s Wrong with APL2

James A. Brown was on the design team for APL2, IBM's successor to APL1. He wrote an article for APL Berlin 2000 [Bro00a] called “What’s Wrong with APL2.”

“It is my personal opinion that performance should be considered [when designing a language] and it should be considered last.” [emphasis in the original] Brown goes on to say: “APL success . . . because it was so productive
to use, not because it was fast . . . I still think APL2 is the most productive programming language.” APL’s slowness is not what’s wrong. Brown lists attributes of languages that he considers to be important:

- usability
- extendibility
- changeability
- programmer productivity
- migratability

Some actual design choices for APL2 that are not optimal are:

- Failure to include a primitive to perform n-wise scan
- Not allowing negative numbers to be the left-hand argument to the replicate primitive
- The APL2 equivalent to the try construct in other languages, in which exceptions are caught. The problem was that most programmers couldn’t use it effectively.
- The failure to include a last primitive that would apply to an array. (APL has a first \( X \) primitive – corresponding to LISP’s car – and a 1 drop \( X \) primitive – corresponding to LISP’s cdr).
- Not including strings as a primitive type. APL offers a character vector and using these may require the programmer in some situations to be aware of the length of the vector, where with strings, programmers are often not length-aware.
- Calling the language APL2. Brown states that the name “VS APL Version 2” would have been better for marketing, as more customers would have upgraded.
- Failure to include a standard library. Without one, “everyone has his or her own ‘standard’ library.”

References


A.D. Falkoff and D. L. Orth, *Development of an APL standard*, International Business Machines, 1979, IBM’s internal standard for APL is the appendix to the paper. Contains a BNF for APL.


Kenneth E. Iverson, *A dictionary of APL*, APL Quote Quad 35 (2007), no. 1–2, 32–67, This is a reprint of the original article which was published in the same journal in 1987.


[Zor09] Denis Zorin, January 2009, Talk at NYU’s Courant Institute on the tools he uses in his research.