A development of *APL2* syntax

by James A. Brown

This paper develops the rules governing the writing of *APL2* expressions and discusses the principles that motivated design decisions.

1. Introduction

IBM has numerous products which follow the IBM internal standard for *APL* (*VSAPL*, *APLSV*, PC *APL*, 8100 *APL*). In this paper this level of the *APL* language is referred to as *APL*1.

*APL*2 is based on this writer's Ph.D. thesis [1], the array theory of Trenchard More [2], and most of all on *APL*1. It incorporates extensions to data structures, to primitive operations, and to syntax. Those wishing a complete description of *APL*2 may refer to the *APL*2 publications library listed in the general references. The only extensions covered here are those which have an effect on the syntax of the language and those implied by the simplifications of syntax.

A presentation of syntax would be brief. This is, rather, a discussion of how the syntax was designed and what motivated the choices that were made. A longer discussion of this and related topics appears in "The Principles of *APL*2" [3].

2. The objectives of *APL2* syntax

*APL*1 has always had a simple syntax governed by only a few rules. These rules are phrased in terms of general statements that are easy to apply in practice: "All functions have equal precedence"; "functions are executed from right to left"; "operators have higher precedence than functions"; etc.

When language extensions are proposed we find that the familiar rules do not cover all cases. For example, *APL*1 has the concept of an operator (for example, /) applying to a function (for example, +) and producing a new function (called summation). In *APL*2 operators are generalized so that they can apply to all functions—including those produced by other operators. Therefore a syntactic decision has to be made about the meaning of a statement like

```
+.×/
```

(where "." is the matrix product operator). This could mean either

```
(+.×)/ or +.(×/)
```

Operators are extended so that they take arrays as operands. Therefore, if "*DOP*" is a dyadic operator taking an array right operand,

```
+ DOP A×B
```

could mean

```
(+ DOP A)×B   or   + DOP (A×B)
```

and the rules of *APL*1 are of no help. In either case × gets evaluated before *DOP*. The only question is "does × have one or two arguments?"

These questions and others like them could have been resolved by stipulating new rules that cover the cases followed by a determination that no syntactic ambiguity was introduced. Instead *APL*2 uses the concept of binding strength, which brings together in one measure all the concepts of syntax—order of execution; precedence of operators over functions; building lists of arrays; etc.

It is well known that any precedence grammar can be described by a matrix [4, 5]. This is made easier in *APL*2 (and even *APL*1) because it has a small number of object classes. Bunda and Gerth [6] give a development of such a matrix as a model for evaluating various extensions to *APL* syntax. However, even a small matrix can be hard to remember and apply in practice. Benkard [7] shows that

*APL*2 can be simplified to a linear hierarchy of syntax classes. It is this concept that is developed here.

3. The objects of *APL*2

*APL*2 recognizes three classes of objects: arrays, functions, and operators. It is this set of objects and the operations defined on them that must be expressed by the syntax. Since the kind of data we want to represent is already known and the style of the syntax is already given, it is not a surprise that choices are made that make the syntax and the universe of objects work together.

Thus we chose arrays that are finite, rectangular collections of other arrays ultimately comprised of numbers and characters. We chose operations (functions and operators) which may take at most two arguments and which therefore can be easily represented by an infix notation.

4. Names

The first step in developing the rules for syntax is establishing the rules for identifying the objects of discourse in the written notation.

A *name* is a string of one or more characters which is, or may be, associated with an *APL*2 object. Some names are always associated with the same object; others may not be associated with objects at all or may be associated with different objects at different times.

Names are considered atomic, indivisible units of writing even when they take more than one character to represent.

Once names are identified, they are considered the tokens of the syntax and their structure is never again of interest.

• *Primitive names*

Primitive names are those that are defined as part of the definition of the language. They have fixed associations in that a given primitive name is always associated with the same object.

Primitive array names

*APL*2 arrays are collections of numbers and characters. The primitive arrays (the ones given names) are single numbers and single characters (that is, simple scalars).

Numeric scalars are written using their decimal representations. Complete rules for writing numbers may be found in [8]. Here are examples of various styles of numbers:

```
245.5        2J3
¯245.5       2D45
2.35E13      1R1.716
```

(The second column shows three ways to write complex numbers.)

While any decimal number may be written, in an implementation not all are associated with a scalar object. For example, `2E987654` is a legal name for a number but is not associated with an object in most implementations because the number is not representable.

A given numeric object may be associated with many names. For example, the number "fifteen" can be written

`15` or `15.0` or `1.5E1` or `15J0`.

Character scalars are written by enclosing the graphic associated with the character in single quotation marks.

`'A'`

This is a single character and is treated as an indivisible unit despite the fact that on input it occupies three print positions. The use of the quotes means it is always possible to distinguish between a number which is represented by a single digit and the character whose graphic is that digit.

Primitive operation names

Primitive operations are named by single symbols each of which occupies one print position.

There is a large set of primitive functions using the symbols

```
+ - × ÷ * | ⌈ ⌊ ? ⍟ ○ ! ⌹ ~ ∧
∨ ⍲ ⍱ < ≤ = ≥ > ≠ ≡ ⍴ ⊃ ⊂ , ⍪ ⌽ ⍋ ⍒
⍳ ∈ ⍷ ⍸ ∈ ⍷ ⍸ ⍲ ⊥ ⊤
```

There are only a few primitive operators using the symbols

```
. / \ ⌿ ⍀ ¨
```

Note that dot (`.`) is an ambiguous symbol used as a decimal point as well as an operator. Which is intended in any instance is determined from context.

• *Constructed names*

Constructed names are strings of one or more characters with the following constraints:

An initial or only character is from the set

```
ABC...XYZΔ⍙
ABC...XYZΔ
```

and remaining characters (if any) are from the set above (excluding quad) along with

```
0123456789¯_
```

User names

User names follow the above rules except that the initial character may not be ⍙. Any name constructed according to these rules is valid (no length limitation), and none has any value (i.e., none is associated with an object) until some action is taken to specify the association. User names may be associated with any class of *APL*2 object.

Arrays and user names are associated through use of the specification arrow (←), through parameter substitution caused by invoking a defined operation, and as an implicit

result of the $\square TF$ function. A name which is associated with an array is called a *variable.*

Functions and operators are associated with user names as an implicit result of the $\square FX$ and $\square TF$ functions. Functions may also be associated with user names through parameter substitution in a defined operator. Thus a user name may indicate the same function as a primitive function or even a derived function.

Distinguished names

Names which begin with the character $\square$ are reserved for fixed uses in the language and are called *distinguished names.* Any distinguished name is valid, but only a few are associated with objects. Distinguished names associated with arrays are called *system variables*; distinguished names associated with functions are called *system functions.*

- *Writing names*

When writing a linear sequence of names, a primitive operation name need never be separated from adjacent names. Thus the two names + and − written next to each other (+ −) can never be confused with a single different operation. All other names may require more than one symbol. When writing a linear sequence of names, these names, if adjacent, must be separated to avoid confusing the combination with a single different name. Thus the two names 1 2 and 3 4, when written next to each other, must be separated to avoid confusion with the name 1 2 3 4. The separation character is a blank if no other nonblank character falls between them. For example,

```
1 2  3 4     blank needed for separation
1 2 ( 3 4 )  blank not needed for separation
```

5. Syntax

This section and following sections show the derivation of the definition of syntax for $APL2$. The Appendix includes a summary of the rules.

The syntax of $APL1$ is simple, straightforward, and easy to learn. This is so because of the great care exercised by the creators of $APL1$. Similar care is required in making any extensions or changes to syntax. With the exception of the removal of mixed output, the syntax has been unchanged since the early days of the language. Therefore extensions to syntax are probably the most constrained by $APL1$. The resulting syntax must retain at least the following properties:

- It is linear—we do not want superscripts, radical signs, and so forth.
- It uses a function symbol for two (usually related) functions—one monadic and one dyadic.
- There is no functional precedence—all functions have equal precedence and execute according to their position in an expression.
- Operators have higher binding power than functions.

The syntax of $APL2$ must be able to express

- Arrays,
- Functions and their application to arguments,
- Operators and their application to operands.

The linear collection of special symbols and names (primitive and constructed) used to write arrays, functions and their application to arguments, and operators and their application to operands is called an *expression.*

The names and symbols used to write an expression are divided into six syntax classes:

- Array,
- Function,
- Monadic operator,
- Dyadic operator,
- Assignment arrow,
- Brackets.

(Note that the object class operator is divided into two syntax classes: Brackets and their contents are treated as one class; ∘ and → are treated like functions.) To these classes are added parentheses—the only punctuation symbols in an expression.

Evaluation of an expression may produce any of the three objects or may produce no object at all and be correct (although an attempt to display or assign the result of an expression that produces a function or operator generates an error).

An expression is classified by the object it produces:

- Array expression: one that evaluates to an array.
- Function expression: one that evaluates to a function.
- Operator expression: one that evaluates to an operator.
- Valueless expression: one that evaluates to no object.

Evaluation of an expression involves scanning the names (in a strictly right to left order), determining binding strengths of objects next to each other, and evaluating operations whenever they are completely determined. Thus the fundamental concept of syntax is that of adjacency or juxtaposition and its use for the most important actions: forming of vectors, applying functions to arguments, and applying operators to operands. An actual model of evaluation using this scheme can be found in [6].

When two names are written next to each other, there is an affinity between them—that is, the combination means something in the notation. This affinity is called *binding strength.* When three names are written next to one another, the middle one exhibits affinities for the names on the left and the right. One of these affinities is stronger than the

other, indicating that that construct in the language is more important. In the following text, we examine binding strengths of various combinations of objects. The goal is to arrive at a simple linear hierarchy that is easy to use in practice to parse expressions. Bindings are chosen so that useful expressions can be written without parentheses. Parentheses are introduced as a way to force one binding when another, stronger one would normally prevail. The stronger binding is delayed while what is inside the parentheses is evaluated. This is called a *delayed binding.*

• *Expressions without parentheses*

First, we investigate how to write arrays, functions, and operators and discover the bindings implied when symbols and names of objects are placed next to each other.

Array expressions

Array expressions are divided into two groups. The first involves the writing of vectors and the second the writing of other array-producing expressions.

Vector array expressions There is one rule for writing a simple vector: Write the simple scalars which are the items of the simple vector next to each other with separating blanks as needed. Since the rule involves a separation of items, the resulting vector must have at least two items.

Here are three examples of simple constant vectors. The first is all numeric, the second is a mixture of numbers and characters, and the third is all character:

```
2 3 4
2 'B' 4
'A' 'B' 'C'
```

The last example is a different way of writing a simple character vector from that provided in *APL*1. (A compatible way of writing a character vector is covered in the discussion of vector expressions in parentheses.)

This is the first extension to syntax and is a simplification. There is now one rule for writing a vector: Write the scalar items separated by spaces. This may be generalized by saying that when two arrays are written next to each other, there is a binding between them. If I and J are arrays, writing them next to each other implies construction of a vector containing them as items. This is called vector binding.

Other array expressions Given that we can write some arrays, we may now consider how we write functions and apply them to arrays. The rule is the same as in *APL*1: A function symbol may represent two functions—one monadic (one argument or valence 1) and one dyadic (two arguments or valence 2). A monadic function is written with its single argument on the right and a dyadic function is written with arguments on the left and the right (infix notation):

monadic function ÷ 2

dyadic function 5 ÷ 2

It could be argued that, if ÷ 2 is a monadic function, then 5 ÷ 2 is the number 5 sitting to the left of a monadic function. This is even easier to argue if instead of ÷ we use a symbol which does not have a dyadic definition. For example, the symbol used for enclose (⊂) has not been given a dyadic meaning. One could argue that 2 ⊂ 3 is really a 2 next to a monadic function. *APL*2 solves this possible ambiguity with the following rule:

> All functions are ambi-valent (both valences) and the one written in any instance is determined only by context.

Thus functions in the abstract are ambi-valent, but at evaluation time (call time) the syntax uniquely determines which function is intended. If one wrote a function symbol with an argument on each side, he would have written a dyadic function. In the case of ⊂, if this should ever be given a dyadic meaning, it would not be considered a change to the syntax of *APL*2—it would be a change to the semantics. This is why in *APL*2 attempting to execute such an expression gives *VALENCE ERROR* rather than *SYNTAX ERROR*.

In the same sense that arrays written next to each other have vector binding, arrays next to functions have argument binding. In the following this is called left argument binding and right argument binding.

When an expression is written containing more than one function, rules for determining which is to be evaluated first must be given. In the expression

```
2×3+4
```

which is done first—the multiplication or the addition? Another way of phrasing this question is "Which gets bound to the 3? × or +?" *APL*1 has always had a scanning rule called the "Right to left rule":

> In an unparenthesized expression without operators, functions are evaluated right to left.

We can get an equivalent rule by declaring that left argument binding is stronger than right argument binding:

Binding strength (strongest on top)
left argument
right argument

Thus, the above expression means 3+4, then 2× the result.

The next question to answer is "Where does vector binding fit in with argument binding?" Beginners in *APL*, not being told otherwise, often assume that vector binding is

lower than right argument binding so that in the expression

```
2×3    4+5    (extra spaces for emphasis),
```

times binds its right argument 3 and plus its left argument 4, getting two results 6 and 9, and that then these are bound giving the two-item vector 6 9. There is absolutely nothing wrong with this analysis except that *APL*2 chooses to put vector binding higher than argument binding. Thus *APL*2 has the following hierarchy:

Binding strength
- vector
- left argument
- right argument

In the above example 3 is bound to 4 first and then the pair is bound to + as its left argument. It is this choice that gives *APL*2 its array processing capabilities. The fundamental data in *APL*2 are arrays. We therefore make it easy to construct arrays and apply functions to them.

Function expressions

Without operators the only function expression that can be written is one which contains only the name of a function. Thus

```
×
```

is a syntactically correct function expression. It means we are talking about the function itself, as opposed to its application to arguments. Therefore the above expression results in the function "times." Although it is an error to attempt to display or assign this result, in the future even this could be allowed and would not be an extension of syntax. Without these extensions, function expressions are useful only in expressions containing operators. The reason for allowing function expressions becomes clear after parentheses are discussed.

Operators can be used to write other function expressions, in which case the function result is called a *derived function.*

The syntax of operators is in many aspects the mirror image of the syntax of functions. A monadic operator is written with its single operand on the left:

```
+/   for   /   a monadic operator
```

A dyadic operator is written with its operands on the left and the right:

```
+.×   for   .   a dyadic operator
```

Each of these evaluates to a derived function and so is a valid function expression. As before, the attempt to display the derived function generates an error.

Operators differ from functions (even in mirror image) in that they have fixed valence. A particular operator is either monadic or dyadic but never both. This is why operators are represented by two syntax classes.

*APL*2 permits the operand of an operator to be any function—even the function which results from the application of another operator. Without a rule, the following expression is ambiguous:

```
+.×/
```

This could be an inner product between + and ×/ or it could be a reduction by an inner product. The question is further complicated by the possibility of array operands.

As with functions the answer can be approached by specifying the binding strengths of operators to their operands. In *APL*1 operators have always been thought of as "more powerful" than functions, and this concept can be turned into an assignment of binding strengths. Since the operands are presented in the mirror image of functions, we choose binding strengths in the mirror image. Thus we stipulate the following:

Binding strength
- right operand
- left operand

with the understanding that monadic operators have no binding strength on the right at all. Therefore the conclusion is that in the expression

```
+.×/
```

the right binding strength of . is stronger than the left binding strength of / and the expression is a reduction by an inner product.

These bindings must now be fitted in with those already determined. Any choice is correct, but the *APL*1 expression

```
A+.×B
```

requires that right operand binding be higher than left argument binding. This gives

Binding strength
- right operand
- left argument
- right argument

Left operand binding could go in any of three places (since it is below right operand binding), but since we are not trying to express the sum of A with anything, we make left operand binding higher than right argument binding. Because no object is both a function and an operator, the ordering of left argument and left operand does not matter. Therefore the binding hierarchy for functions and operators is defined as

Binding strength
- right operand
- left operand
- left argument
- right argument

It is in this sense that operators have higher precedence than functions; they have stronger bindings.

Now only vector binding needs to be placed in the hierarchy. Because / is an operator, compatibility with *APL*1 requires that vector binding be higher than left operand binding. In the expression

```
1 0 1/A
```

we want the vector to be formed before the left operand of / is bound. Therefore vector binding must be stronger than left operand binding, leaving two possibilities:

Binding strength

```
          <- - - -
  right operand
          <- - - -
  left operand
  left argument
  right argument
```

Either of these positions is correct, and both were tried experimentally in the *APL*2 Installed User Program [9] (which did not allow array left operands). The question is exemplified by the following expression using a dyadic defined operator *DOP* (there is no primitive dyadic operator that takes an array right operand):

```
+ DOP A B
```

If vector binding is above right operand binding, this is a function expression with *A B* as the right operand. If vector binding is below right operand, this is an array expression which applies the derived function + *DOP A* to argument *B*. This second choice makes operators with array right operands easy to use and so is the order chosen.

Therefore the binding hierarchy for functions, operators, and vectors is

Binding strength

- right operand
- vector
- left operand
- left argument
- right argument

Operator expressions

The only operator expressions are a single operator name or a single operator name to the left of brackets. (Brackets are discussed separately.)

Valueless expressions

User-defined functions that do not return explicit results may be written. The only valueless expressions that can be written involve such a user-defined function, the primitive function execute (⍎) whose evaluation includes such a function, or an empty expression such as

```
L1: ⍎ EMPTY EXPRESSION
```

• *Expressions with parentheses*

In *APL*1 parentheses are used only to group functions with their arguments. In *APL*2 there is the need to express other groupings (for example, grouping an operator with its operands). Rather than use a new pair of grouping symbols, a new simplified parentheses rule has been adopted. This rule is

> Parentheses are used for grouping.

They may be used anywhere as long as they are properly paired and what is inside the pair evaluates to an array, a function, or an operator. An expression inside parentheses (or one which could be put in parentheses without changing the evaluation of anything) is called a *subexpression.*

Evaluating expressions with parentheses is only a matter of evaluating what is inside the parentheses and then substituting for the parenthesized expression the value it produces. This leads to a statement of a substitution rule that is the basis for mechanical evaluation of *APL*2 expressions [3].

Correct parentheses that do not delay any bindings are called *redundant parentheses* and may be removed from, or added to, an expression without affecting the result of the expression. While this is a sufficient definition of redundant parentheses, it is useful to identify particular cases where parentheses are needed.

Parentheses surrounding a single name or an expression already in parentheses could not delay any bindings and so must be redundant. For example,

`2 (+) 3`	Constant operation name
`A+(.)×B`	Constant operation name
`(A)←3`	Constructed name
`(2)+1`	Constant array name
`((2 - 3))+1`	Parenthesized expression

Here is an example of parentheses that seem redundant by this rule but are not:

`(NDFN)` niladic function without result

These parentheses are not correct (let alone redundant) because what is inside does not evaluate to any array, a function, or an operator.

Array expressions with parentheses

Again, array expressions are divided into two groups.

Vector array expressions in parentheses In expressions of arrays, parentheses that do not separate a group from another part of the expression are redundant. For example,

```
2 (3) 4      These do not group.
(2 3 4)      These group but do not separate.
```

Nonredundant uses of parentheses in vector expressions give a facility for writing nested vectors. For example, consider

```
2 (3 4)
```

What is inside the parentheses is a valid *APL2* expression and so the parentheses are correct. Evaluating what is inside the parentheses gives us an array (a two-item vector). Vector binding tells us that writing 2 next to an array gives us a vector. Thus parentheses may be used to write nested vectors. This is called *vector notation* in *APL2* and *strand notation* by others [2].

The following rewriting rule provides *APL1* compatibility 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.

The parentheses must be part of the rule even though they appear redundant. Thus in the following example even though `'B' 'C'` is made up entirely of single characters, the rewriting rule may not be applied:

`'A' 'B' 'C'` is not `'A' 'BC'`

The following is a correct application of the rule:

`('A' 'B' 'C')` is rewritten	`('ABC')`	Rewriting rule
`('ABC')` is rewritten	`'ABC'`	Remove redundant parentheses

Other array expressions in parentheses Parentheses in array expressions are redundant if they group the right argument of a function or a vector left argument of a function:

```
2×(3÷4)      Group right argument.
(2 3)×4      Group vector left argument.
```

Function expressions in parentheses
Parentheses in function expressions are redundant if they group the left operand of an operator:

```
(+.×)/       Group left operand.
```

Parentheses around a function expression are redundant if the left parenthesis does not separate two arrays:

```
A (+.×) B    Group function expression.
```

However, the following parentheses are not redundant because the left parenthesis separates arrays:

```
A (B/) C     Nonredundant parentheses.
```

Operator expressions in parentheses
It is not possible to write an operator expression that uses nonredundant parentheses. Even in an operator expression involving brackets, parentheses are redundant. (Brackets are discussed separately.) Thus, in any syntactically valid operator expression, parentheses are redundant.

Valueless expressions in parentheses
A valueless expression may not be a subexpression (that is, may not be within parentheses). Writing a valueless expression in parentheses results in a *VALUE ERROR*.

Brackets are a special syntactic construction for writing lists of arrays for use in indexing and axis specifications. They are correct if correctly paired and if what is inside is one of the following:

- Nothing `[ ]`
- An array expression `[1] [2+2]`
- More than one of the above separated by semicolons `[;] [1;] [1;2 3 4]`

Brackets are used for two different purposes: indexing and axis specification. In each case evaluating a bracket expression is a substitution in that brackets to the right of an array (indexing) produce an array, and brackets to the right of a function or operator (axis specification) yield a function or operator, respectively.

- *Indexing*

Brackets indicate an indexing function when written to the right of an array expression (a single name or an expression in parentheses):

```
A[2]
(matrix expression)[3;]
```

Such constructions are always syntactically correct, but there are domain restrictions implied by the semantics of brackets. Namely, the rank of the array indexed must equal 1 plus the number of semicolons inside the brackets. The consequences of this are that brackets cannot be used to index a scalar and cannot be used to the right of an expression that at different times produces an array of different rank.

- *Axis specification*

Brackets indicate an axis specification when written to the right of a function or operator expression (a single name or an expression in parentheses):

```
⌽[1]
```

The brackets are considered to be a notation for writing an operation related to the one on its left. It cannot be considered an operator because the definition of the related function cannot be expressed, in a uniform way, in terms of the original function.

Writing the brackets next to a function or operator is always syntactically correct, but evaluation of the related function or operator succeeds only under specific conditions. An *AXIS ERROR* is generated when the conditions are not met. The conditions are as follows:

- The bracket expression must contain no semicolons.
- If the related function is used monadically, the original function must be one of ⊃ ⊂ , ⍎ ⍕.
- If the related function is used dyadically, the original function must be one of ⍎ ⍕ , ↑ ↓ and the scalar functions.
- If the related operator is monadic, the original operator must be one of / \ ⌿ ⍀.

The primitive functions mentioned above may be written as primitive symbols or as user names having the primitive operation as value (because of parameter substitution in a defined operator).

Here are examples of incorrect axis specifications:

`2↑[2;3]A`	semicolons in brackets
`⍳[3]A` and `⍴⍨[1]A`	⍳ and ⍨ not allowed

The reason why the brackets are not treated as applying to the derived function ⍴⍨ is presented in the next section.

Evaluation of the related function could yield many error conditions including *AXIS ERROR* for other reasons. For example,

```
⍎[5] 2 3 4
```

is allowed by the conditions but gives an *AXIS ERROR* because 5 does not indicate an axis of the argument array.

- *Binding strength*

Brackets are not an array, a function, or an operator. They are treated as members of a special syntactic class. We must, therefore, make an individual assessment of where they fall in the binding hierarchy. The following example shows that there is a choice. Let *DOP* be a dyadic operator:

```
+ DOP ⍎ [1]
```

If right operand binding is higher than bracket binding, this must mean

```
(+ DOP ⍎)[1]
```

which gives an *AXIS ERROR* because the rules do not include any valid use of brackets with a derived function. If bracket binding is higher than right operand binding, this must mean

```
+ DOP (⍎[1])
```

which is a legal function expression. Neither choice is more formally correct. The second option lets us write a useful expression without parentheses and is the option chosen in *APL2*. As usual, parentheses may be used to delay binding, but no useful expression can be so produced.

If brackets have stronger binding than right operands then, if we are to maintain the simple linear hierarchy, their binding is stronger than any other binding yet discussed, giving the following hierarchy:

Binding strength
- brackets
- right operand
- vector
- left operand
- left argument
- right argument

This implies that in the expression

```
+/[1]A
```

the brackets bind to the operator / producing a new monadic operator which binds to + as its left operand.

A useful way to phrase the binding strength of brackets is to say that "Brackets are tightly bound to the object on their left." For example,

```
A +.×[2] B
```

expresses an inner product with operands + and ×[2]. If *A*, *B*, and *C* are vector arrays, then

```
A[1] B[2] C[3]
```

expresses the three-item vector whose first item is *A*[1], whose second item is *B*[2], and whose third item is *C*[3].

```
A B C[2]
```

is a three-item vector whose first item is *A*, whose second item is *B*, and whose third item is *C*[2]. Substituting scalar integers for *A*, *B*, and *C* in the above example shows that

```
2 3 4[2] ↔ 2 3 (4[2])
```

which is a *RANK ERROR*. Such constant vectors are viewed as expressions containing the names of three scalars. This is different from *APL1*. Indexing of a constant numeric vector requires parentheses. [Note that) *MCOPY* (Migration *COPY*) and) *IN* make this change in defined functions migrated from *APL1* [8].]

The practical effect of this placement of brackets in the hierarchy is that brackets become syntactically transparent. Whenever brackets are seen in an expression (for indexing or

axis specification), they bind tightly to whatever is on the left and the combination may be immediately evaluated and replaced by the computed value from the same class. This is why brackets and their contents may be treated as a single syntax class. Parentheses around brackets and the object to their left do not delay any bindings and are always redundant.

Brackets, which have always been an exceptional case in *APL*1 (sometimes described as a function and sometimes as an operator), are now regularized and explained.

7. Other special symbols

*APL*2 includes the use of several special symbols that do not represent arrays, functions, or operators. These are parentheses, brackets, semicolons, right and left arrows, and jot. Parentheses, brackets, and semicolons have been treated previously.

• *Assignment*

The assignment arrow (←) is the only syntactic construction for associating names with arrays. There are two kinds of assignment: one which associates a name (perhaps with no value) with an arbitrary array (*direct assignment*); and one which merges an array into indicated positions in another array already associated with a name (*selective assignment*). In each case one parameter is an array and the other is either a name or positions in a named array. Therefore the assignment arrow can be neither a function nor an operator (since these operate on values, not names). The assignment arrow is in a separate syntactic class.

The name whose value is replaced or modified must be a constructed name having no value or having an array value. This, in particular, excludes names of niladic defined functions which are otherwise treated syntactically as arrays.

Assignment syntax

To fit assignment into the binding hierarchy, we must consider the relative strengths with which a left arrow binds with what is on its left and what is on its right. *APL*1 answers both these questions.

Consider the expression

```
A←2+3
```

Clearly, left argument binding must be stronger than assignment right binding so that the addition is done before the assignment. Assignment right binding must therefore be placed either just above or just below right argument binding. Because the left arrow cannot be a function, the order is immaterial. We therefore elect to place assignment right binding as lowest, giving the following binding hierarchy:

Binding strength
- brackets
- right operand
- vector
- left operand
- left argument
- right argument
- assignment right

*APL*1 only helps a little in determining assignment left binding. The expression

```
2+A←3
```

shows that assignment left binding is stronger than right argument binding. Because *APL*1 did not have operators with array operands, we may choose how much stronger than right argument binding it is.

Consider the following expression, where *DOP* is a dyadic operator with array right operand:

```
+ DOP A←3
```

If right operand binding is stronger than assignment left binding, then this means

```
(+ DOP A)←3
```

which is an error. If instead assignment left binding is stronger than right operand binding, this means

```
+ DOP (A←3)
```

which is a legal function expression. This is the choice made in *APL*2, giving the hierarchy

Binding strength
- brackets
- assignment left
- right operand
- vector
- left operand
- left argument
- right argument
- assignment right

(Because brackets do not bind on the right at all, assignment left could have been put at the top.)

This choice of assignment left binding has the practical effect of tight binding a left arrow to the thing on its left. Thus an assignment can always be immediately evaluated and replaced by its value (which is always the array on its right), making assignments syntactically transparent.

Assignment result

While assignment is not treated like a function, it may be thought of as a function whose explicit result is the value of its right argument. Alternatively it may be considered

syntactically transparent in the sense that after the assignment is complete, the arrow and whatever is bound to it on the left are removed from the expression, leaving the right argument array as value. In either case, after the assignment, a value is left and is considered the explicit result of the assignment. This may then be used in further computation.

Here are some examples of assignments in value expressions and the value that is computed:

Expression	Value after execution
`A←3`	3
`(A←3)`	3
`(A←2),(B←3)`	2 3
`2+A←1`	3
`(A←2) (B←4)`	2 4

The following rule determines when the value of an expression should be displayed:

> If the last syntactical action in a value expression is an assignment, the final array value of the expression is not displayed. If any binding occurs after the last assignment, or if there is no assignment, the final array value is displayed.

Here are executions of the above examples using this rule:

```
A←3
          no display—last action is assignment
(A←3)
          no display—last action is assignment
                     parentheses are redundant
(A←2),(B←3)
2 3       display—last action is binding of 2 and 3 to
                     catenate (followed by execution)
2+A←1
3         display—last action is binding of 2 and 1 to
                     plus (followed by execution)
(A←2) (B←4)
2 4       display—last action is binding 2 to 4
                     (no function executed)
```

• *Branch and escape*

The right arrow, when used to control sequencing in a defined operation or when used to resume execution, is called *branch*. It is syntactically like a function and so does not influence the binding hierarchy. It fails to be a function in the strict sense because it does not have an explicit result and is not ambi-valent (dyadic use gives *SYNTAX ERROR*). It can therefore only be used in a valueless expression. The execute function (⍎) and user-defined operations may also fail to return an explicit result but are nonetheless still considered functions. Branch is not considered a function semantically and in particular cannot be the operand of an operator. Its only purpose is the determination of the next line to be executed.

When the right arrow is used without a right argument, it is called *escape*, and it must be the only symbol in the expression. Syntax is not a question because nothing is next to it.

• *Jot*

The jot symbol "∘" is used as a special symbol to distinguish between the two derived functions of the array product operator dot (.). If the left operand of matrix product is a function (*F* . *G*), the derived function is *inner product*. If the left operand of matrix product is jot (∘ . *G*), the derived function is *outer product*. *Inner product* (*F* . *G*) takes two functions as operands. *Outer product* (∘ . *G*) takes one function as operand, and the jot is a place holder for the other operand. Its use is not exploited or extended beyond its use in *APL*1.

Strictly speaking, jot is in its own syntactic class. Syntactically, however, it is treated as a function when it is used in the context of *outer product* and so does not influence the binding hierarchy. It cannot be used as an operand to other operators, but expanding its use would introduce no formal problems.

The preceding derivation of the rules of *APL*2 syntax can be summarized in a few pages (see the Appendix). The derivation of the rules is seen as the orderly investigation of the usefulness of written expressions as influenced by a few general principles. Binding gives one concept that ties together the concepts of order of execution, precedence of operators over functions, use of parentheses, etc. The principles can be phrased in terms of a few simple rules that are easy to apply in practice, with the general rule always ready to mediate any apparent ambiguities.

In addition to providing a simplified view of *APL*2 syntax, the principles give a framework under which other extensions to *APL*2 can be considered.

The definition of *APL*2 evolved over a fifteen-year period during which many people contributed thoughts and suggestions. Most notably I wish to thank Garth Foster, who was my advisor when many of these ideas were forming; Trenchard More, whose work prompted the use of vector notation in *APL*2; Adin Falkoff, who together with Ken Iverson developed the syntax for operators and generally supported the project for over a decade; and Phil Benkard, who promoted the concept of binding. "The Principles of *APL*2" [3] contains a list of many other people who contributed to the definition and implementation of *APL*2.

10. Appendix: A summary of APL syntax

Object classes There are three classes of objects:

- arrays
- functions
- operators

Function valence All functions are ambi-valent (both valences) and the one written in any instance is determined only by context.

Operator valence Operators have fixed valence. A given operator is either monadic or dyadic, determined by definition, not context.

Syntax classes There are six syntax classes:

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

Parentheses rule Parentheses are used for grouping. They are correct if properly paired and if what is inside evaluates to an array, a function, or an operator.

Redundant parentheses Correct parentheses that do not alter any bindings are redundant:

- general
  - group a single name (primitive or constructed)
  - group an expression in parentheses
- array expressions
  - do not both group and separate
  - group right argument of a function
  - group vector left argument of a function
- function expressions
  - group left operand of an operator
  - group function expression and left parenthesis does not separate two arrays
- bracket expressions
  - group brackets and object to the left

Expression A linear string of names and symbols, taken from the six syntax classes, punctuated with parentheses.

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

Function precedence Functions in an expression have no precedence. The order of execution depends only on position in the expression.

Rewriting 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.

Printing results If the last syntactical action in a value expression is an assignment, the final array value of the expression is not printed. If any binding occurs after the last assignment, or if there is no assignment, the final array value is printed.

Binding hierarchy

brackets
assignment left
right operand
vector
left operand
left argument
right argument
assignment right

brackets	binding of brackets to what is on the left
assignment left	binding of a left arrow to what is on its left
right operand	binding of a dyadic operator to its operand on the right
vector	binding of an array to an array
left operand	binding of an operator to what is on its left
left argument	binding a function to its left argument
right argument	binding of a function to its right argument
assignment right	binding of 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 that produces no value.
Niladic functions are syntactically arrays.

1. J. A. Brown, "A Generalization of *APL*," Doctoral Thesis, Dept. of Computing and Information Science, Syracuse University, Syracuse, New York, 1971.
2. T. More, "Notes on the Development of a Theory of Arrays," *IBM Philadelphia Scientific Center Report 320-3016*, May 1973.
3. J. A. Brown, "The Principles of *APL* 2," *IBM Santa Teresa Technical Report TR 03-247*, March 1984.
4. R. W. Floyd, "Syntactic Analysis and Operator Precedence," *J. ACM* **10**, No. 3, 316–333 (1963).
5. V. R. Pratt, "Top Down Operator Precedence," *Conference Record*, ACM Symposium on Principles of Programming Languages, 1973.
6. J. D. Bunda and J. A. Gerth, "*APL* Two by Two—Syntax Analysis by Pairwise Reduction," *Proceedings of APL84*, Helsinki, Finland, 1984, pp. 85–94.

7. J. P. Benkard, "Valence and Precedence in *APL* Extensions," *APL Quote Quad* **13**, No. 3, 233–242 (1983).
8. *APL2 Programming: Language Reference*, Order No. SH20-9227, 1984; available through IBM branch offices.
9. *APL2 Installed User Program: Language Manual*, Order No. SB21-3015, 1982; available through IBM branch offices.

The following are general references to *APL2* and *APL* extensions.

APL2 General Information, Order No. GH20-9214, 1984; available through IBM branch offices.

APL2 Migration Guide, Order No. GH20-9215, 1984; available through IBM branch offices.

APL2 Programming: System Services Reference, Order No. SH20-9218, 1984; available through IBM branch offices.

APL2 Programming: Using Structured Query Language (SQL), Order No. SH20-9217, 1984; available through IBM branch offices.

J. A. Brown, "Evaluating Extensions to *APL*," *APL Quote Quad* **9**, No. 4, Part 1, 148–155 (June 1979).

J. A. Brown, "*APL* Syntax—Is it Really Right to Left," *APL Quote Quad* **13**, No. 3, 219–221 (1983).

A. D. Falkoff and K. E. Iverson, "*APL*\360 User's Manual," Order No. FH20-0683-1, 1970; available through IBM branch offices.

A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth, "A Formal Description of SYSTEM/360," *IBM Syst. J.* **3**, Nos. 2/3, 198–263 (1964).

Ziad Ghandour and Jorge Mezei, "General Arrays, Operators and Functions," *IBM J. Res. Develop.* **17**, No. 4, 335–352 (1973).

K. E. Iverson, "Operators and Functions," *Research Report RC-7091*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1978.

K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, 1962.

R. H. Lathwell and J. E. Mezei, "A Formal Description of *APL*," *Colloque APL*, Institut de Recherche d'Informatique et d'Automatique, Rocquencourt, France, 1971.

T. More, "Notes on the Axioms for a Theory of Arrays," *IBM Philadelphia Scientific Center Report 320-3017*, May 1973.

Trenchard More, Jr., "Axioms and Theorems for a Theory of Arrays," *IBM J. Res. Develop.* **17**, No. 2, 135–175 (1973).

T. More, "A Theory of Arrays with Applications to Databases," *IBM Cambridge Scientific Center Report G320-2107*, September 1975.

T. More, "Types and Prototypes in a Theory of Arrays," *IBM Cambridge Scientific Center Report G320-2112*, May 1976.

T. More, "On the Composition of Array-Theoretic Operations," *IBM Cambridge Scientific Center Report G320-2113*, May 1976.

T. More, "Notes on the Diagrams, Logic, and Operations of Array Theory," *IBM Cambridge Scientific Center Report G320-2137*, September 1981.

Received April 13, 1984; revised June 4, 1984

IBM General Products Division, P.O. Box 50020, San Jose, California 95150. Dr. Brown graduated from Syracuse University, New York, in 1971 with a Ph.D. in computer and engineering science. He then joined the *APL* design group at IBM Research in Yorktown Heights, New York. He also worked at the Philadelphia Scientific Center, at the Palo Alto Development Center, and then at the Santa Teresa laboratory. Dr. Brown is currently manager of *APL* language development at the Santa Teresa laboratory. He is language editor for the *APL Quote Quad*. The recently announced *APL2* program product is based on his doctoral thesis, "A Generalization of *APL2*."