APL2: Getting started

by J. A. Brown
H. P. Crowder

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

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

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

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

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

APL2 arrays

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

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

```
      A
3

      NUMS
1 3 5 7 3.14159

      CHAR
CAT
FAT

      MIXED
2 BE OR NOT 2 BE
```

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

```
      DISPLAY A
3

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

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

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

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

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

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

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

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

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

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

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

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

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

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

APL2 functions

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

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

```
      3ρ'A'
AAA

      2 2ρ1 2 3 4
1 2
3 4

vs      2 3ρ'CATFAT'
CAT
FAT
```

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

```
      ρ3ρ'A'
3

      C
CAT
FAT

      ρC
2 3
```

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

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

```
      ρρ2 3ρ'CATFAT'
2
```

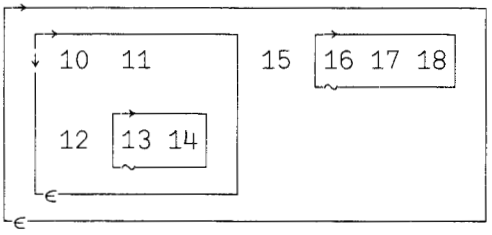
```
      ⍴⍴1 2 3
1

      ⍴⍴'A'
0
```

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

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

      1 2 3 ≤ 3 2 1
1 1 0

      100 × 1 2 3
100 200 300
```

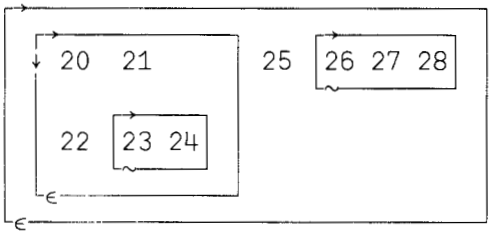
and in the example before of a nested array *D*, where:

```
      DISPLAY D
```



an arithmetic function example is:

```
      DISPLAY D+10
```



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

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

```
      A
1 2 3

      B
100 200 300

      A,B
1 2 3 100 200 300

      M
1 2
3 4

      N
100 200 300
400 500 600

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

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

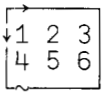
Interval (denoted by ⍳) produces arrays based on numerical sequences. **Interval** and arithmetic scalar functions can be combined to produce a wide variety of arrays:

```
      ⍳7
1 2 3 4 5 6 7

      2×⍳7
2 4 6 8 10 12 14

      ¯8+2×⍳7
¯6 ¯4 ¯2 0 2 4 6

      2⋆⍳7
2 4 8 16 32 64 128
```

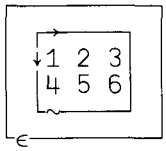
Notice that in APL2 expressions, functions are executed from right to left.

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

```
      DISPLAY Q
```



```
      DISPLAY ⊂Q
```

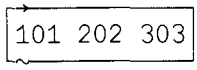


In the second expression above, the *DISPLAY* function shows the result to be a scalar. The data are organized along no axes and have rank 0. Inside the scalar, however, the complete original array is retained. Therefore **enclose** returns a scalar that contains its argument as its only item.

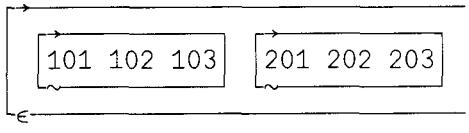
This data structure has several practical applications. Arrays are sometimes used in situations where the structure is not important. **Enclose** allows hiding the inner structure of arrays. For example, *'JIM'* is a three item character vector. If an application treats this array as a name then the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

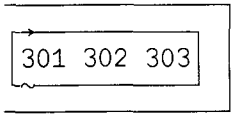
Enclose is also useful if the contents of an array are required to participate in scalar extension. Note the difference that **enclose** makes in the following examples:

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




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





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

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

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

       AVG 3 9 7 11 14
8.8

       2 × AVG 3 9 7 11 14
17.6
```

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

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

       SD 3 9 7 11 14
3.709
```

APL2 operators

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

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

```
       +/1 3 4 2 5
15

       ⌈/1 3 4 2 5

5
```

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

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

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

      +/M
10 26 42

      ⌈/M
4 8 12
```

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

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

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

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

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

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

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

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

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

      R←2 3 ⍴ 'ABCDEF'
      DISPLAY (⊂3 4) ⍴¨R
┌→──────────────────────────┐
↓ ┌→──┐ ┌→──┐ ┌→──┐          │
 ↓AAA│ ↓BBB│ ↓CCC│          │
  AAA│  BBB│  CCC│          │
  AAA│  BBB│  CCC│          │
 └──┘ └──┘ └──┘            │
  ┌→──┐ ┌→──┐ ┌→──┐         │
  ↓DDD│ ↓EEE│ ↓FFF│         │
   DDD│  EEE│  FFF│         │
   DDD│  EEE│  FFF│         │
  └──┘ └──┘ └──┘           │
└∈──────────────────────────┘
```

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

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

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

      AVG¨ A
5  3  6.25
```

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

APL2 examples

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

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

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

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

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

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

```
2 CIRCLE 0 0
```

Figure 1A shows the resulting picture.

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

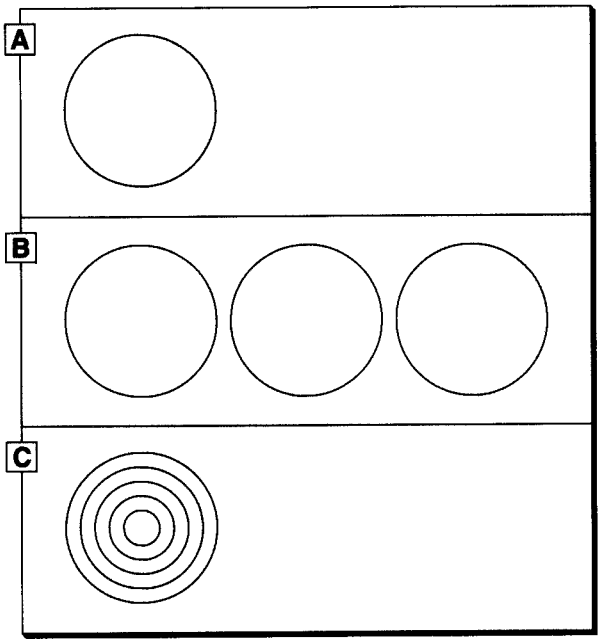
```
2 CIRCLE¨ ¯4 0 4,¨0
```

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

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

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

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




```
(⍳5) CIRCLE¨ ⊂0 0
```

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

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

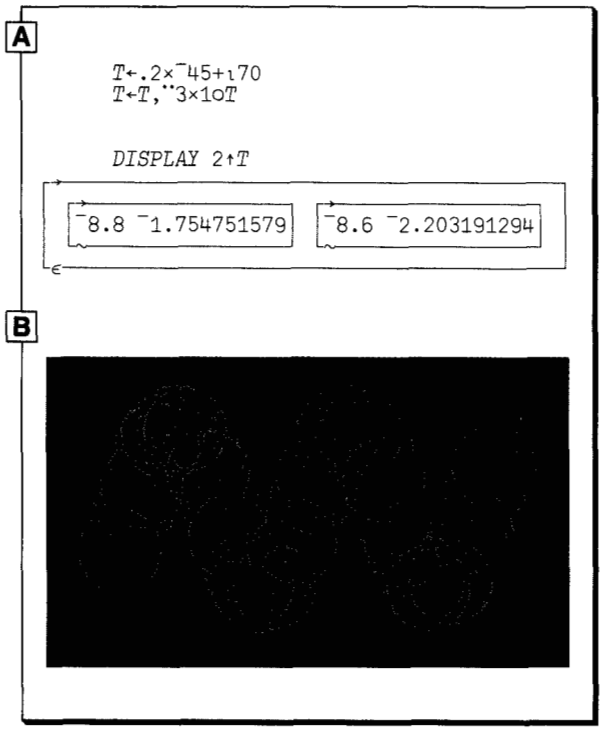
```
T←.2×¯45+⍳70
T←T,¨3×1○T
```

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

```
1.5 CIRCLE¨ T
```

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

Figure 2 Result of drawing multiple circles



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

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

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

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

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

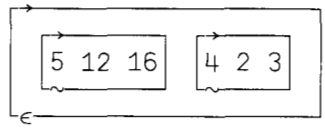
```
[0]   Z←SVPACK V;I
[1]   ⍝ PACKS A FULL VECTOR <V>
[2]   ⍝ INTO A SPARSE VECTOR <Z>
[3]   I←V≠0
[4]   Z←(I/⍳⍴V)(I/V)
```

Now the result of:

```
      SV←SVPACK V
```

is:

```
      DISPLAY SV
```



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

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

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

      FV SIP SV
30

      FV +.× V
30
```

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

```
      ⍴SM
4
```

```
      ρ¨SM
2 2 2 2
```

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

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

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

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

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

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

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

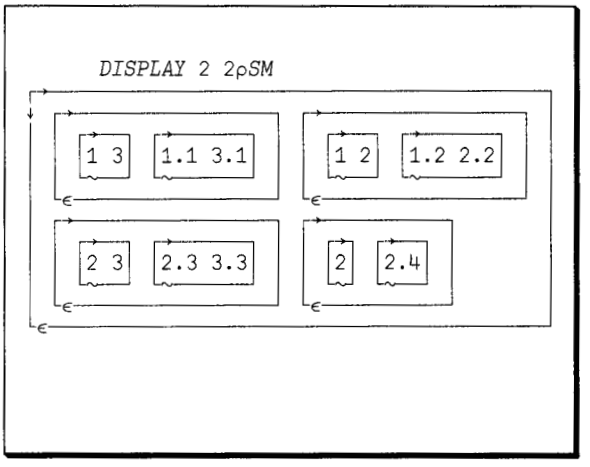
The function *DICE* is used to simulate a prescribed number of rolls of a pair of dice:

```
[0]   Z←DICE N
[1]  ⍝ ROLL DICE <N> TIMES
[2]   Z←?(N,2)ρ6
```

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

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

Figure 3 A two-by-two matrix




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

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

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

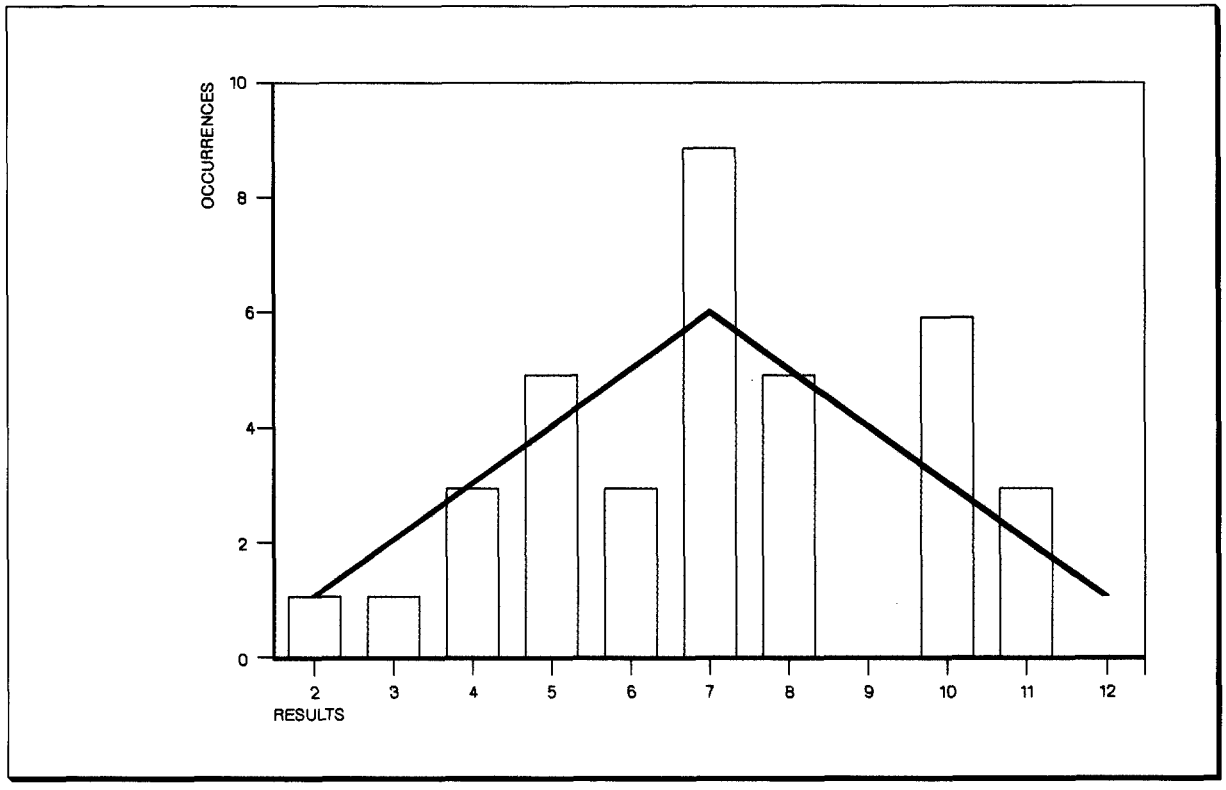
```
[0]   Z←COUNT A
[1]  ⍝ COUNTS DICE THROWS IN <A>
[2]   Z←+/A
[3]   Z←+/(1+⍳11)∘.=Z
```

Now the expression:

```
      A←DICE 7
```

results in:

Figure 4 Result of 36 dice throws




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

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

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

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

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

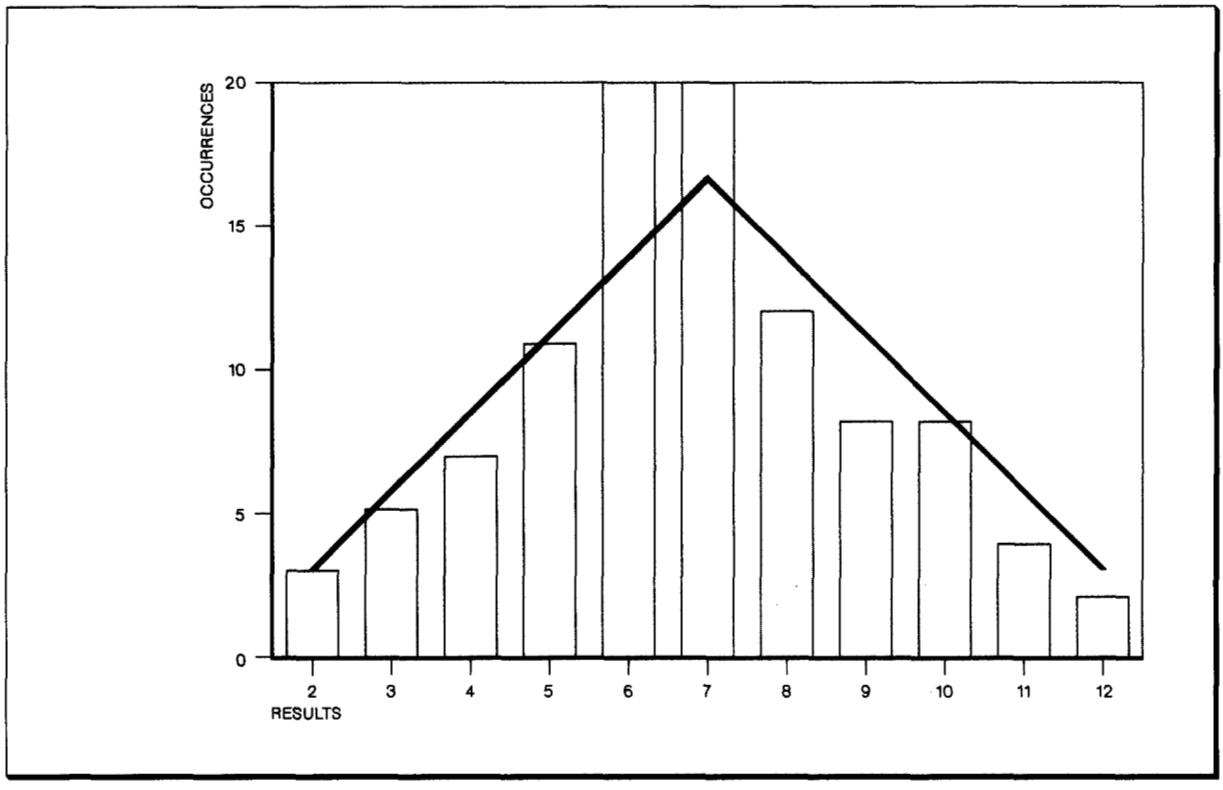
Continuing the discussion, the function *EXPECT* can be used to compute the expected number of dice-pair sums for a prescribed number of rolls:

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

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

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

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

Figure 5 Result of 100 dice throws




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

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

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

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

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

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

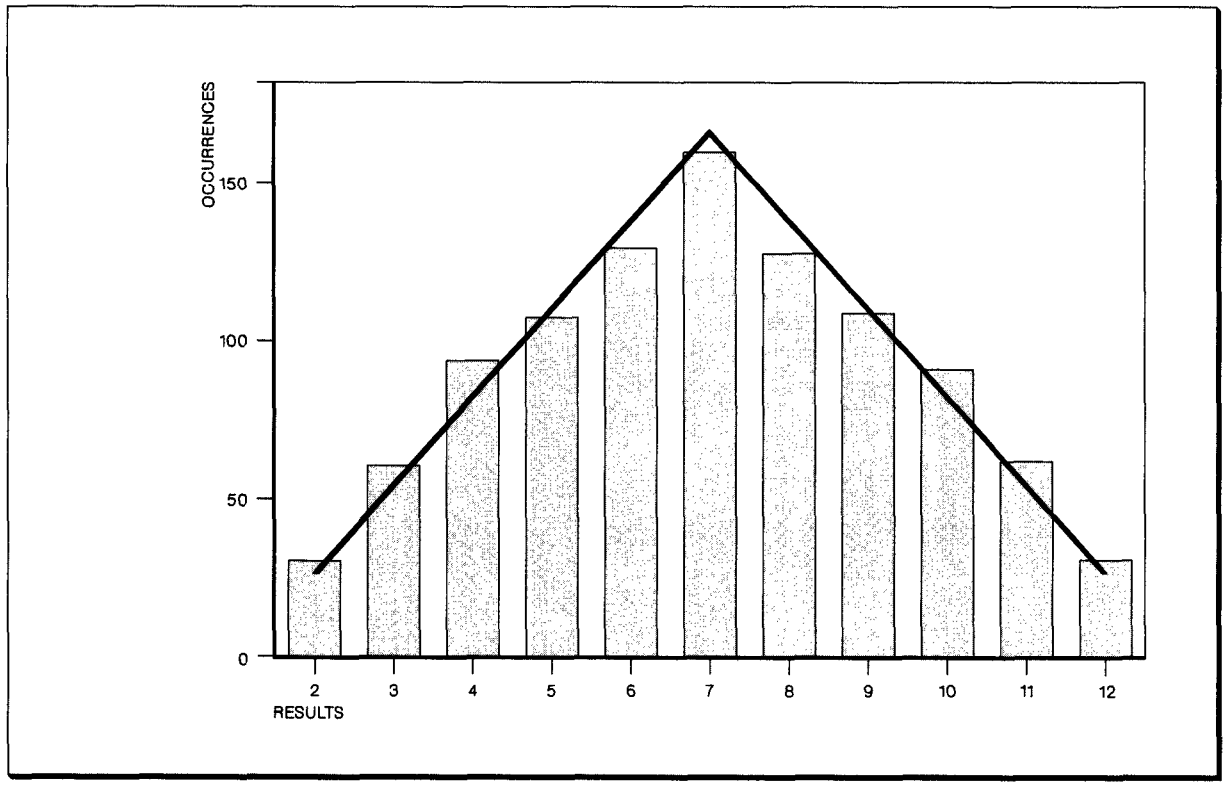
```
DRAW (COUNT DICE 100) (EXPECT 100)
```

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

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

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

Figure 6 Result of 1000 dice throws



Conclusion

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

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

*Trademark or registered trademark of International Business Machines Corporation.

Cited references and note

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

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

Accepted for publication June 21, 1991.

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

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

Reprint Order No. G321-5444.