FAC: A Functional APL Language

Hai-Chen Tu, GTE Laboratories, and
Alan J. Perlis, Yale University

FAC, a functional array calculator language, features APL syntax and array operations but allows partitions, operators, and infinite arrays—extensions that are semantically implausible in APL.

In the last two decades, the development of programming has unveiled new, important language concepts. In most cases, each concept is characterized by a programming language and is then identified as the paradigm of the language. Once the paradigm of a language has been recognized, the language is expanded accordingly. In the meantime, a language must also be ready to adopt other paradigms because new tasks emerge and its own paradigm may not be able to handle them. When two or more paradigms are integrated, each must contribute its strengths without weakening the others. In this respect, APL integrates nicely with functional languages.

This article describes the functional array calculator language FAC and gives programming examples. Details of implementation, however, are described elsewhere.[1]

Historical review

Functional languages and list processing. Functional (applicative) programming creates shorter and more compact programs than ordinary algorithmic (imperative) programming. It abstracts a problem through a sequence of function compositions instead of state transitions. The major difference is that functional programming maintains referential transparency while algorithmic programming does not. Referential transparency means that any variable or expression, within a given scope, can associate only with a single value. Without referential transparency, a program is harder to write and to debug, which makes program maintenance difficult and thus contributes to the software crisis.

Most of the work described in this article was done while Hai-Chen Tu was at Yale University.

A shorter version of this article appears in *Conf. Record HICSS-19,* Hawaii International Conference on System Sciences, January 8-10, Honolulu.

The Bettmann Archive

One of the most delightful of all imaginary species, the centaur of Greek mythology, symbolizes the feeling of oneness between man and horse. The first centaurs were often thought to be offspring of Ixion, a king of Thessaly, and a cloud that Zeus had made in the shape of a goddess. Anoher theory is that centaurs were the offspring of a Greek hero with a serpent's tail, Centaurus, and the mares of Magnesium. But, since horse-riding was unknown to Greeks of the Homeric period, it may be that the first Scythian horsemen they met in battle really seemed to be dual creatures of the centaur kind.

0740-7459/86/0100/0036$01.00 © 1986 IEEE

List processing

Almost all functional languages have adopted lists as their main data structure because lists have a simple structure and flexibility. Even though list processing is orthogonal to functional languages, there are good reasons that functional languages rely on lists. First, since lists can simulate any desired data structures, there is little need for other data structures. Second, list processing fits quite well with recursive programming, which is strongly advocated by most functional languages. Finally, the simple semantics of list structure (just rely on car, cdr, and cons) can be easily implemented in any functional language. All these contribute to the misconception that functional programming is the same as Lisp-style list programming.

From the programming point of view, Lisp-style list processing has some drawbacks in at least two respects. Lists may not be the appropriate data structure to represent problems, and the sequentiality introduced by its structure and recursion may not reflect the parallelism one could have. Lisp-style list processing thus creates the intellectual von Neumann bottleneck in functional languages. Users have to think and program in terms of scalar operations. Here is a simple example. The problem is to decide whether or not a set of numbers are all nonzero. The Lisp solution is to represent the set as a linear list and check through the list recursively for any zero.

```
(define (check S)
   (cond ((null S) t)
      ((= 0 (car S)) nil)
      (t (check (cdr S)))))
(check '(1 2 3 0 4)) ≡ nil
(check '(1 2 3 4)) ≡ t
```

(In Lisp, *nil* represents the Boolean value false, and *t*, true.)

The Lisp solution is not as compact as the predicate shown below using set notation. The predicate also exhibits a high degree of parallelism, which the Lisp solution lacks.

$\forall i \in S; i \neq 0$

Not only do functional languages exhibit better programming style, but they also show more parallelism than nonfunctional languages: arguments of a function application can be executed in parallel because there are no side effects. Indeed, both the dissatisfaction with sequential languages and advancing hardware technology have generated the research boom on dataflow machines—computers that can execute functional programs efficiently.[2]

Unfortunately, the parallelism and compactness of functional programs haven't been fully realized because Lisp-style list processing has dominated functional languages (see box above).

To search for a better functional programming style, we can either investigate new styles of list processing or choose some other data structures. The FP system proposed by Backus[3] is an attempt of the first kind; it retains the list structure (streams) but advocates nonrecursive functional programming style by using functional forms. On the other hand, we see little progress in identifying alternative data structures other than the current trend of research on typed lambda calculus,[4,5] which emphasizes structure programming more than breaking the von Neumann bottleneck. However, we feel that a good candidate for an alternative data structure—arrays—has been effective in a nonfunctional language, APL. (Although some Lisps contain arrays, none has extended the array processing capability close to what APL has.)

APL and array processing. The central idea of APL programming is to treat arrays, instead of their individual elements, as basic data units. The concept of manipulating arrays as single units creates a unique style of programming in which computation is composed through a series of array transformations. Thus, APL enables us to write very condensed programs and to exhibit highly parallel computation.

The powerful APL programming style is not derived from Algol-style, scalar-oriented controls, which are actually a distraction, but from linear algebra notation. More specifically, the style uses partitions (for example, $A+B$ distributes $+$ over scalar elements of A and B), and operators (for example, $+/A$ sums all A's elements). The former explores parallelism, and the latter abstracts controls; both contribute to the compactness of APL programs. And, of course, we must credit the uniform structure of arrays for making these notations possible. In all, we can say arrays, partitions, and operators form the backbone of APL programming style. Brief explanations of some APL terminologies are given in the box on p. 39. Array index origin 0 is used throughout this article.

The APL solution for the check-no-zeros problem (see box above left for the description of the problem) is given below. Note the solution is the same as the predicate shown before, except the notations are different.

```
∧/(0≠S)
∧/(0≠1 2 3 0 4) ≡ ∧/1 1 1 0 1 ≡ 0
∧/(0≠1 2 3 4) ≡ ∧/1 1 1 1 ≡ 1
```

The three APL constructs have been well recognized, and in the last decade, the most significant progress in APL has been in generalizing those constructs. For example, in APL2,[6] nested arrays are used, partitions are extended to defined functions using the each (¨) operator, and operators become user definable (see box on p. 38).

But all extensions fall short of expectation because of a major weakness in APL—its semantics. APL was initially designed as an algorithmic language with properties like gotos and side effects. Also, since it was designed as an interactive language, it uses dynamic binding instead of the preferred lexical binding. (When APL was designed, nobody knew how to implement an interactive lexical binding

APL extensions and their problems

Each operator. The each (¨) operator allows a function to be applied in parallel to array elements. For example, a user can make his own scalar functions, which work similarly to primitive scalar functions. For example,

```
SQ I : I×I
SQ 3 ≡ 9
SQ ¨ 1 2 3 ≡ (SQ 1), (SQ 2), (SQ 3) ≡ 1 4 9
```

Although the implementation of such parallelism is not our concern, we do not expect the order of evaluation of the each operation to affect the final result. The following example, however, shows that it does affect the result when the side effect is involved.

```
A← 1
INC X: A← A+X
B← INC ¨ 1 1 1
```

The function inc has one parameter *X*. When invoked, it increases the global variable *A* by *X* and returns the new value *A*. Vector *B* is the result of applying the function inc in parallel, using the each operator, to the vector 1 1 1. Since *A* is changed each time inc is applied, we have a problem deciding the exact values of *B* and *A*. If the computation is executed in parallel, *B* should become 2 2 2, and *A*, 2. If it is executed in sequence, then *A* may become 4 and *B* may become 2 3 4, 4 3 2, or any permutation of 2, 3, and 4. We say "may" because we haven't considered the effect of any possible optimization. For example, if an optimizer has decided that only the second element of *B* will be accessed, it can compute that element only. In this case, the second element is 2, and *A* becomes 2.

Defined operators. In many occasions, we found no adequate APL operator to express an abstract operation. An extension that allows users to define their own operators will certainly make APL more flexible. However, the extension may produce an equally undesirable result (as that of the each operator) not only because of side effects, but also because of dynamic binding.

Dynamic binding means that the binding between a variable and a memory cell is done at runtime. The variable may bind to different memory cells at different times. (The opposite is lexical binding, which binds a variable to a fixed place, which is defined by its lexical position.) The problem with dynamic binding is that we may bind a variable to an undesirable place, without user awareness, and produce an incorrect result. The funarg problem is an example of dynamic binding intervened with defined operators.

For example, we can define an operator apply whose only operation is applying its operand to its argument.

```
(G APPLY) X: G X
```

This operator appears to be part of a harmless operation, and (*F* APPLY *Y*) should be equivalent to (*F Y*) for any *F* and *Y*. But the following counterexample gives us a different view of the operator.

```
X ← 1
F Y : X + Y
F APPLY 2 ≡ 4
F 2 ≡ 3
```

(*F* APPLY 2) first binds *G* to *T* and *X* to 2. When *G* is applied with the value 2, the *X* inside function *F* binds to the X of apply, not to the global variable *X*. So *G* 2 produces 4, and is not equal to *F* 2, which is 3.

language.) Extending APL mathematical features without rethinking APL's semantics makes all extensions questionable and the whole language ambiguous.

For example, the each (¨) operator in APL2 cannot be treated as a parallel operator because the function it applies to may contain side effects. Or in defined operators, the funarg problem caused by dynamic binding severely handicaps their usage.

FAC: functional array (APL) calculator language

We have shown that both functional languages and APL have their strengths and weaknesses, and that combining them seems to enhance the strengths and eliminate the weaknesses. These benefits have led us to design and implement a functional APL language—FAC—to do experiments.

The syntax of FAC is pretty much the same as that of APL, but FAC has functional semantics. That is, FAC uses lexical binding and allows no side effects or gotos. Moreover, both conditionals and recursions are undefined in this language because they are not part of the APL programming style. We use the name calculator to show that the power of APL programming is in constructing expressions, not programs. We expect FAC to be integrated with other functional languages, not to be treated as a standalone product.

FAC objects. There are three kinds of objects in FAC: arrays, functions, and operators. From a functional language point of view, they are all functions. Arrays are functions that map natural numbers to values. Operators are second-order functions that map functions to functions. Since each plays a different role in FAC, we prefer to treat them as three different classes of objects instead of just calling them functions.

Constants. A constant definition binds a global name to an expression that computes to an array.

```
TEN : 10
PI : 3.1415
TWOPI : PI×2
```

A constant definition can be thought of as a niladic function (a function that takes no argument) that always evaluates to the same value because there are no side effects.

Operator-derived functions. A function can be monadic, dyadic, or ambivalent.[7] (An ambivalent function is actually a set of two functions, one monadic and the other dyadic, of the same name. When applied, depending on the number of arguments [one or two], an appropriate function from the two is selected and applied.) Functions are right associative and follow the rules of long-right scope; that is, the right argument of a function is everything to its right. Parentheses must be used for different function-argument grouping. Assume F is a function and A is an array, then

F F A ≡ F (F A)
F A F A ≡ F (A F A)

The definition below defines a dyadic function plus.

X PLUS Y : X + Y

Ambivalent functions may be defined. Below is the definition of an ambivalent function sqsum.

L SQSUM R : (L×L)+(R×R)
SQSUM R : (R×R)
3 SQSUM 4 ≡ (3×3)+(4×4) ≡ 25
SQSUM 5 ≡ (5×5) ≡ 25

Operators. Operators are restricted second-order functions; they take and return functions. Like functions, operators can be monadic, dyadic, or ambivalent. In a monadic operation, the operand is placed to the left of the operator. Operators are associated to the left and follow the rule of long-left scope. That is, the left operand of an operator is the longest subexpression, extended from the left of the operator, that derives to a function. Also, operators have higher precedence than functions.

F O F O ≡ (F O F) O
F O O F O F O ≡ (((F O) O F) O F) O

where O is an operator and F is a function.

The apply operator defined below is a monadic operator and derives to a monadic function.

(F APPLY) R : F R
(− APPLY) 1 ≡ ¯1

Derived functions can also be ambivalent. Iverson's[8] dyadic operator del (∇) derives to an ambivalent function.

L (F ∇ G) R : L F R
(F ∇ G) R : G R
5 − ∇ + 2 ≡ 5−2 ≡ 3
− ∇ + 2 ≡ +2 ≡ 2

The swap operator swaps the left and right arguments before applying the function operand.

L (F SWAP) R : R F L
1 −SWAP 2 ≡ 2−1 ≡ 1

The APL reduction and scan operators are extended to a family of operators.

Some APL notation and primitives

These simplified definitions are for those APL notations and primitives used here, many of which are explained for scalar and vector arguments only. A complete list of definitions can be found in any APL introductory book.

Assignments. A← ~ 1 assigns the value 1 to the variable A.
Negative constants. In APL, ¯2 stands for the negative constant "negative 2." While −2 means "negate the constant 2," and it returns the number ¯2.
Booleans. In APL, 1 and 0 stand for the Boolean values true and false, respectively.
Array index references. An array index operation can take in array of indices, instead of a single index as in Fortran.
4 5 6[0] ≡ 4
4 5 6[2] ≡ 6
4 5 6[2 1 0] ≡ 6 5 4

Monadic functions

+, −, . . . (unary arithmetic functions) ::
+ 1 2 3 ≡ 1 2 3
− 1 2 3 ≡ ¯1 ¯2 ¯3
~ (*not*) :: the unary Boolean operation
~1 ≡ 0
⍳ (*index*) :: generate an index vector
⍳N ≡ 0 1 2 . . . (N−1)
⍴ (*shape*) :: number of elements of the argument vector
⍴1 2 3 4 ≡ 4
⌽ (*reverse*) :: reverse the argument vector
⌽1 2 3 4 ≡ 4 3 2 1

Dyadic functions

+, −, . . . (binary arithmetic functions) ::
1 2 3 + 1 ≡ 2 3 4
1 2 3 + 4 5 6 ≡ 5 7 9
⍳ (*index-of*) :: the first index in a vector V where a given number is found, else return ⍴V
0 1 2 3 ⍳ 3 ≡ 3
0 1 2 3 ⍳ 8 ≡ 4
⍴ (*reshape*) :: N⍴V duplicates the vector V repeatedly to make a vector of length N
5 ⍴ 1 2 ≡ 1 2 1 2 1
2 ⍴ 1 2 3 4 5 ≡ 1 2
⌊ (*minimum*) :: the smaller one of two given numbers
0 ⌊ 1 ≡ 1 ⌊ 0 ≡ 0
(*maximum*) :: the larger one of two given numbers
0 ⌈ 1 ≡ 1 ⌈ 0 ≡ 1
, (*concate*) :: concate two vectors into a new one
1 2 3 , 4 5 6 ≡ 1 2 3 4 5 6
↑ (*take*) :: take a prefix/postfix subvector
2 ↑ 1 2 3 4 5 ≡ 1 2
¯2 ↑ 1 2 3 4 5 ≡ 4 5
↓ (*drop*) :: drop a prefix/postfix subvector
2 ↓ 1 2 3 4 5 ≡ 3 4 5
¯2 ↓ 1 2 3 4 5 ≡ 1 2 3
=, ≠, ≥, ≤, >, < (comparisons) :: compare two scalars
2 = 2 ≡ 1
2 = 4 ≡ 0
∧, ∨, ⍲, ⍱ (Booleans) :: binary Boolean functions
1 ⍲ 1 = ~ (1 ∧ 1) = ~ 1 = 0
1 ⍱ 0 = ~ (1 ∨ 0) = ~ 1 = 0

Operators (*f* is a scalar function, and V is a vector)

f⌿V (*reduction*) ::
+⌿1 2 3 4 ≡ 1+(2+(3+4)) ≡ 10
−⌿1 2 3 4 ≡ 1−(2−(3−4)) ≡ ¯2
f⍀V (*scan*) ::
−⍀1 2 3 4 ≡ 1, (1−2), (1−(2−3)), (1−(2−(3−4))) ≡ 1 ¯1 2 ¯2

1. *right-reduction, reduction* (⌿, or /):
   F⌿1 2 3 4 ≡ 1 F (2 F (3 F 4))
2. *left-reduction* (⍀):
   F⍀1 2 3 4 ≡ ((1 F 2) F 3) F 4
3. *right-scan, scan* (⍋, or \):
   F⍋ 1 2 3 4 ≡ 1, (1 F 2), (1 F (2 F 3)), (1 (2 F (3 F 4)))
4. *left-scan* (⍒):
   F⍒ 1 2 3 4 ≡ (((1 F 2) F 3) F 4), ((2 F 3) F 4), (3 F 4), 4
5. *right-propagation* (⍿):
   F⍿1 2 3 4 ≡ (1 F (2 F (3 F 4))), (2 F (3 F 4)), (3 F 4), 4
6. *left-propagation* (⍀):
   F⍀1 2 3 4 ≡ 1, (1 F 2), ((1 F 2) F 3), (((1 F 2) F 3) F 4)

It is possible to let the scan and reduction operators simulate the rest of these operators, but explicit definitions make abstracting solutions much easier.

Function forms. Besides operator-derived functions, other syntax forms create local functions.

Functions can be curried. (*V F*) denotes a curried function, where *F* must be dyadic. The parentheses are necessary to ensure nonambiguous syntax. Obviously, a curried function is always monadic.

(V F) ⇔ λR.F(V,R)
(1 +) 2 ≡ 3

(The equivalent lambda expression is shown to the right of ⇔.) The curry notation permits only the left argument to be curried. To curry the right argument, we need the operator swap.

(1 −SWAP) ⇔ λZ. (λLR. R − L)1 Z ≡ λZ.Z − 1
(1 −SWAP)2 ≡ 2 − 1 ≡ 1

There is another way to create local functions—αω-forms. An αω-form is an anonymous function enclosed inside a pair of curly brackets; it is similar to a lambda form in lambda calculus, but it uses two default parameters—α and ω (for the left and right parameter, respectively)—instead of naming parameters explicitly.

{α+ω} ⇔ λαω. α+ω
5 {α+ω} 1 ≡ 6

Nested αω-forms are allowed but not encouraged because they may confuse users.

1{ω{α−ω}α}2 ≡ 2{α−ω}1 ≡ 2−1 ≡ 1

An αω-form is dyadic when α is present. If a dyadic αω-form is applied in a monadic case, FAC automatically binds the parameter α to the value *nil*. In general, a domain error will result unless the αω-form doesn't access the value of α (for example, sequential and [∧]). An αω-form is ambivalent if the parameter α is missing.

{α+ω} 1 ≡ nil+1 ≡ error
{ω∧α} 0 ≡0∧nil ≡ 0
{2+ω} ⇔ λαω. 2+ω
{2+ω} 1 ≡ 3
4 {2+ω} 1 ≡ 3

Curried functions can be substituted for by αω-forms. (1 +) is the same as {1+ω} except that the latter is ambivalent, not monadic.

FAC assignments. An assignment in FAC is not a side-effect action; it creates a new lambda binding that binds a local variable to an object. All classes of objects (arrays, functions, and operations) are assignable.

A←10 ⇔ (λA.A) 10
1+A← 10 ⇔ (λA.1+A) 10
1(A← +) 2 ⇔ (λA.A(1,2)) +

Assignment is right associative and has the lowest precedence.

A←B←10 ≡A←(B←10)
A← +10 ≡ A←(+ 10) not (A← +)10

The scope of a local variable starts from where the name is assigned to the left end of the whole expression. If the assignment is inside an αω-form, the scope is not extended outside the αω-form.

1{A←α+ω}2 ⇔ (λαω.(λA.A)α+ω)1 2 ≡ 3
A+{A←ω+A}A←1 ⇔
(λA.A+((λαω.(λA.A)(ω+A))nil A)) 1 ≡ 3

Note that *A* : 10 is different from *A* ← 10. The former defines a global constant *A* with the value 10; the latter creates a local variable *A* with the value 10 and returns the value *A*, which is 10.

The powerful griffin spends most of its time repelling attacks of the greedy Arimaspians, a race of one-eyed people who direct their ingenuity and effort toward stealing the griffins' carefully hoarded gold.

The Bettman Archive

Multiple assignments to the same name within an expression are allowed and create nested environments.

```
1+A←2+A←3 ⇔ (λA.(λA.1+A) (2 + A))3
(A←1)(A←+)(A←2) ⇔ (λA.(λA R.(λA F.F(A,R))1 A)+ A)2
```

In the lambda expression of the last example, two local variables *R* and *F* are needed to hold the values of the right argument, and the function before the left argument is evaluated.

Arrays. FAC arrays are an extension of Lowney's carrier arrays,[9] but they support more operations and are extended to infinite arrays. Since the scope of this article is limited, we do not present a formal array definition based on denotational semantics. Such a definition can be found elsewhere.[1]

Ragged arrays and partitions. The internal structure of a carrier array is a ragged array. A ragged array is a vector of (ragged) subarrays, where all subarrays must have the same dimensions but not necessarily the same shape. It can be viewed as a uniform depth tree—a tree in which all paths from root to leaves have the same depth, and the depth of the tree defines the rank of the ragged array.

```
A ≡ 1 2 3
    4 5

    6 7 8 9
```

A is a three-dimensional ragged array consisting of two ragged matrices. The first matrix contains two rows—vectors 1 2 3 and 4 5. The second matrix has a single row—the vector 6 7 8 9.

Since ragged arrays are not required to be rectangular, we can distribute a rank-uniform function in parallel to all fixed-rank subarrays of a ragged array, and reconstruct all returned arrays into a new ragged array. A rank-uniform function is a function that returns arrays whose ranks are solely dependent on ranks of arguments. All FAC primitive functions are rank-uniform functions. For example, since the iota (ι) function takes a scalar and returns a vector,

```
ι2 1 4 ≡ ι2 ≡ 0 1
          ι1   0
          ι4   0 1 2 3
```

Rank-uniform functions are a special case of partition operations in which a partition operation directs how to break arguments into subarrays and imposes a rank on returned arrays. A partition has the format *L F*[*z*;*l*;*r*] *R*, which indicates the following operations: (1) break right argument *R* into rank *r* subarrays; (2) break left argument *L* into rank *l* subarrays (this operation is not executed in a monadic function); (3) apply function *F* in parallel to all pairs of subarrays and check each function call return as an array of rank *z*; and (4) collect all returned arrays into a new ragged array. For example the iota function has default partition [1;;0], and the take (↑) function, [1;0;1].

```
L↑R ≡ L ↑[1;0;1] R
1 2 3 ↑ 1 2 3 4 5 ≡ 1↑(1 2 3 4 5) ≡ 1
      6 7 8 9      2↑(6 7 8 9)     6 7
      1 2 3 4      3↑(1 2 3 4)     1 2 3
```

Users can declare a partition for a defined function, which greatly improves the usefulness of partitions. For example, we can declare the partition for an idiom remdup to be [1;;1], which specifies that remdup take a vector and return a new vector. This idiom removes all duplicated items from a given vector.

```
REMDUP V :[1;;1] ((VιV)=φV)/V
REMDUP V ≡ REMDUP[1;;1] V
REMDUP 1 2 3 2 ≡   REMDUP 1 2 3 2 ≡ 1 2 3
               4 5 5         REMDUP 4 5 5     4 5
               4 5 5         REMDUP 4 5 5     4 5
```

Note that, in APL2,[6] we can apply a partition but not prebind it with a defined function, while in Lowney's carrier arrays we can prebind a partition only with a defined function, but cannot apply it explicitly.

Carrier arrays and datums. A carrier array is a ragged array that allows for one layer of nested structure—each array element can be an (enclosed) ragged array. If the array is nested, then every enclosed array element must be of the same rank. An enclosed array element is called a datum and its rank is called datumrank. Enclose (⊂) and disclose (⊃) operations are two primitive operations that change the datumrank of a carrier array. The left argument of the enclose function is the extra rank added to datums of the right argument array, while the left argument of the disclose function is the new datumrank of the right argument. The former function is a relative datumrank adjust operation, while the latter is an absolute one. (A boxed [sub]array denotes a single datum. No boxes are used for scalar [datumrank 0] arrays.)

```
A ≡ 1 2 3 2
    4 5 5
    4 5 5
0⊂A ≡  A
1⊂A ≡ ┌─────┐
      │1 2 3 2│
      └─────┘
      ┌─────┐
      │4 5 5│
      └─────┘
      ┌─────┐
      │4 5 5│
      └─────┘
2⊂A ≡  1⊂(1⊂A) ≡ ┌─────┐
                  │1 2 3 2│
                  │4 5 5  │
                  │4 5 5  │
                  └─────┘
⊃A ≡   0⊃A ≡A
N⊃M ≡ N⊂(⊃M)
```

Most APL primitive functions are now extended to treat datums as scalars. For example, the equal (=) function now compares two datums of the same datumrank. We can thus

extend the idiom remdup to remove duplicate rows from a matrix.

```
REMDUP  (1⊂A)
REMDUP┌1 2 3 2┐≡  ┌1 2 3 2┐
      ├4 5 5┤    ├4 5 5┤
      └4 5 5┘
```

Since the explicit uses of enclosing and disclosing arrays are rather ad hoc, FAC allows users to declare datumranks for defined functions (called function-datumrank).

```
REMDUP V (1;;1): ((V⍳V)=⍴V)/V
REMDUP [N] ≡REMDUP(N⊂A)
```

In this example, the notation (1;;1)* specifies that the idiom remdup can take a carrier array of arbitrary datumrank datums and that it returns another carrier array of the same datumrank datums. Only 1's and 0's are allowed in a function-datumrank declaration. A 1 specifies that the corresponding argument (or returned value) can be a carrier array of nonzero datumrank; otherwise it must be a plain ragged array, and all arrays specified by 1 must have the same datumrank. When the function is applied in the form *LF*[*N*]*R*, it will first enclose both arguments properly before the function is invoked. It will also check the returned array to make sure it has the correct datumrank.

Putting partition and function-datumrank together, the idiom remdup now has the form

```
REMDUP V (1;;1):[1;;1] ((V⍳V)=⍴V)/V
```

Infinite arrays and bottoms. FAC allows arrays to be infinitely large: an infinite array is an array in which at least one of its dimensions is infinitely long. The simplest example is a one-dimensional array with infinite elements. Infinite arrays in APL were first studied by McDonnell and Shallit.[10] Unlike our study with FAC, their study is limited to rectangular arrays, not ragged arrays.

To be able to create and process infinite arrays, we have to add infinity (∞) to the number system. Some primitives are naturally extended to take ∞ as an argument to create infinite arrays.

```
A←⍳∞ ≡0 1 2 3 4 5 ... ≡A[I]=I
B←⍳∞⍴1 2≡1 2 1 2 1 2... ≡B[I]=(1 2)[2|I]
```

Many computations with infinite arrays are not different from those with finite arrays. For example, adding two infinite vectors creates a new infinite vector where each element is defined as

```
(A+B)[I]=(A[I]+B[I])
```

Since an infinite array represents a nonterminating process, we are able to do an unbound search over the entire array. APL array processing is thus no longer limited to for-loop problems but is extended to while-loop ones.

*This notation is different from that of a partition; a partition uses semicolon-bracket form, while a function-datumrank uses semicolon-parenthesis form.

Consider the problem of finding the smallest nonnegative integer X that satisfies the inequality $X^2-2X-5\geq 0$. The APL solution first generates an infinite vector containing all values of X^2-2X-5 (X ranges from 0 to ∞) then returns the index of the first positive number.

```
(0≤V←¯5+(¯2×X)+X×X←⍳∞)⍳1
X← ⍳∞ ≡ 0 1 2 3 4 5 . . .
V←¯5+(¯2×X)+X×X ≡ ¯5 ¯6 ¯5 ¯2 3 10 . . .
0≤V ≡ 0 0 0 0 1 1 . . .
(0≤V) ⍳1 ≡ 4
```

FAC has an iter (⍣) operator that generates infinite vectors sequentially. This operator derives to an ambivalent function.

```
(F⍣) A ≡ A, (F A), (F (F A)), (F (F (F A))), . . .
  W: F⍣A ↔ W [0]=A ∧ W[I+1]=(F W[I])
A (F⍣) B≡ A,B,(A F B),(B F(A F B)),
  ((A F B)F(B F(A F B))), . . .
        W: A F⍣B ↔ W[0]=A ∧W[1]=B∧
          W[I+2]=(W[I] F W[I+1])
{ω+1}⍣0 ≡ ⍳∞ ≡ 0 1 2 3 . . .
1+⍣I ≡ 1 1 2 3 5 8 . . .  (the Fibonacci sequence)
```

The default partition and the function-datumrank for the iter operator are [1;0;0] and (1;1;1), respectively.

Iter operation is the APL style of recursion. It abstracts the concept of looping into a simple form and hides the sequential control from users. It also forces users to separate

The harpy—deceptively mild in appearance here—is actually a wild and furious Greek bird-goddess that flies across storm-darkened skies, screaming shrilly into the wind. It then whirls earthward to steal food right from under people's noses. Trailing filth and an evil stench, it is a disgusting creature that spreads famine, pestilence, and death across a once-fertile land.

Natural Science Picture Sourcebook, Janet Evans, ed., Van Nostrand and Reinhold, 1984

looping and exit conditions when developing programs. This eliminates the explicit synchronization required in von Neumann-style languages and reduces the code complexity. (More examples using the iter operator are shown later.)

Infinite arrays may create array elements that take forever to compute. These elements are called bottoms ($\perp_B$).*

ONES ≡ 1 1 1 1 . . .
$0 \in$ ONES ≡ $\perp_B$

A FAC array is a lazy data-structure[1] where any array element is computed by demand. It means that each index access $A[I]$ behaves as a function call $A(I)$, not as a memory access. The lazy structure allows an array to have bottoms as its elements, while the array itself is not a bottom.

$A \leftarrow (1\ 0 \in ONES) \rightarrow (A[0]=1) \wedge (A[1]=\perp_B) \wedge (\rho A=2)$

Note that our definition is different from Backus' FP language,[3] where a stream is defined as a bottom if any of its element is a bottom.

FAC arrays have another property, an out-of-range index reference returns the element *nil*, instead of domain error. This definition has the advantage that we can use *nil* as the fill element in carrier arrays.

A : 1 2
3↑A ≡ A[0],A[1],A[2] ≡ 1 2 nil

Another advantage is that we now can distinguish bottoms and *nil*s. For example, reverse (ϕ), an infinite vector, is a terminating process.

ϕONES ≡ ONES[∞],ONES[∞ − 1], ONES[∞ − 2], . . .
≡ nil nil nil . . . (*not* $\perp_B$)
(*Note:* $\rho(\phi ONES) \equiv \infty$)

Infinite arrays vs. infinite lists. Infinite arrays and infinite lists[11] differ in the way they are constructed. An infinite list is created by lazy cons and recursion. For example, the following OnesList function creates an infinite list consisting of nothing but 1's.

OnesList = cons(1,OnesList) ≡ (1 . (1 . (1 . (. . .))))

Since recursion is a sequential process, access of the *i*th element in an infinite list requires that all elements be processed before that one. So the sequential data structure only encourages sequential processing of its elements. Infinite arrays use function mapping to create the data structure. Thus, accessing its elements is conceptually parallel.

ONES : $\infty\rho 1$ ≡ 1 1 1 1 . . . ⇔ $\lambda i.1$

Unlike infinite arrays, it is not possible to compute the length of an infinite list, nor to reverse it.

length(OnesList) = $\perp_B$ *compare to* (ρONES) = ∞
reverse(OnesList) = $\perp_B$ *compare to* (ϕONES) = nil nil nil . . .

*We use the subscripted symbol $\perp_B$ instead of the conventional ⊥, since the latter is used in APL as the decode function.

Programming examples

This section presents FAC programming examples, including two algorithms to generate prime numbers, both of which produce an infinite vector of primes in ascending order; a lucky numbers algorithm; and a method of finding the prefix of powers of two.

Primes by test. The simplest way to generate all prime numbers is to select those numbers that pass the prime test; that is, they are greater than 1 and have no exact divisors other than 1 and themselves.

isPrime N :[0;;0] 2 = +/0 = (1 + ⍳N) | N
N ≡ 9
1 + ⍳N ≡ 1 2 3 4 5 6 7 8 9
T ← (1 + ⍳N) | N ≡ 0 1 0 1 4 3 2 1 0
0 = T ≡ 1 1 1 0 0 0 0 0 1
+/0 = T ≡ 3 (*the number of exact divisors*)
2 = +/0 = T ≡ 0 → *9 is not a prime*

Applying isprime to all natural numbers returns an infinite Boolean vector. Compress (/) compresses the Boolean vector over all natural numbers and picks out all primes.

PRIMES ← (isPrime ⍳∞)/⍳∞
⍳∞ ≡ 0 1 2 3 4 5 6 7 8. . .
isPrime ⍳∞ ≡ 0 0 1 1 0 1 0 1 0 . . .
0 0 1 1 0 1 0 1 0 . . . /0 1 2 3 4 5 6 7 8 . . . ≡ 2 3 5 7 . . .

Sieve of Eratosthenes. A more efficient algorithm for generating prime numbers is the method known as the Sieve of Eratosthenes. The algorithm begins with the sequence containing all natural numbers starting from 2. It pops out the first number, which is a prime, from the sequence and filters out all multipliers of the popped number in the sequence, thus generating a new infinite sequence. If we repeatedly apply the pop-and-filter process to keep generating new sequences, then the number popped from each sequence is a prime and is the next largest prime to the last one popped.

The filter operation is a commonly used APL idiom—a test and compress operation—similar to the one we used to generate primes previously.

FILTER V :[1;;1] (0 ≠ V[0] | V)/V
V ≡ 2 3 4 5 6 7 8 9 10 . . .
V[0] | V ≡ 0 1 0 1 0 1 0 1 0 . . .
0 ≠ V[0] | V ≡ 0 1 0 1 0 1 0 1 . . .
(0 ≠ V[0] | V)/V ≡ 3 5 7 9 . . .

Applying the iterate (⍣) operation to the filter function over 2 3 4 . . . results in a matrix in which each row is a filtered infinite vector of the previous row. The first column of the matrix contains all prime numbers.

PRIMES : ,1↑⊃(FILTERS⍣[1]2 + ⍳∞)
$V_0 \leftarrow 2 + \iota\infty \equiv$ 2 3 4 5 6 7 8 9 10 . . .
$V_1 \leftarrow$ FILTER $V_0 \equiv$ 3 5 7 9 . . .
$V_2 \leftarrow$ FILTER $V_1 \equiv$ 5 7 11 . . .

```
. . .
M←FILTER⍥[1]V0 ≡ [V0 V1 V2] . . . ≡ [2 3 4 5 6 7 8 9 . . .]
                                    [3 5 7 9 . . .]
                                    [5 7 . . .]
                                      . . .
,1↑⊃M ≡ 2 3 5 7 11 13 17 19 23 . . .
```

Lucky numbers. The definition of lucky numbers is similar to that of primes obtained by the Sieve of Eratosthenes. It starts with the same sequence 2 3 4 . . . and uses the same pop-and-filter mechanism, but instead of filtering multiples of the popped element, it filters out elements whose positions are multiples of the popped element. For example, if the first element of a sequence is 2, then every element in the sequence whose position index is a multiple of 2 is removed.

The filter function lfilter for lucky numbers is a variation of the function filter for primes.

```
LFILTER V:[1;;1] (0≠V[0]|⍳∞)/V
LFILTER 2 3 4 5 6 7 8 9 10 . . . ≡ 3 5 7 9 11 13 . . .
LFILTER 3 5 7 9 11 13 15 . . . ≡ 5 7 11 13 17 19 23 . . .
LFILTER 5 7 11 13 17 19 23 . . . ≡ 7 11 13 17 23 . . .
```

By replacing the function filter by lfilter in the definition of primes, we have the definition for luckies.

```
LUCKIES : ,1↑⊃(LFILTER⍥[1]2+⍳∞)
M←LFILTER⍥[1]2+⍳∞ ≡ [2 3 4 5 6 7 8 9 . . .]
                    [3 5 7 9 . . .]
                    [5 7 . . .]
                    [7 . . .]
                      . . . .
,1↑⊃M ≡ 2 3 5 7 11 13 17 23 25 . . .
```

Prefix of powers of two. It is known that for any positive integer Y, there exists at least one nonnegative integer X such that Y is a prefix of 2^X. Our problem is to write a function pptwo that returns the smallest X for any given Y. For example, given $Y=5$, the powers of two that start with a 5 are 512 ($=2^9$), 524288 ($=2^{19}$), etc. The solution of these is 512; that is, pptwo 5 ≡9.

We first divide the problem into two parts: (1) generate all powers of two and (2) match Y against all numbers generated by (1) and return the smallest index where a match is found.

Since most powers of two are too large to fit in machine integers, we represent them as bignums, where bigmum is the vector of digits that is the decimal representation of a number. In our case, we are concerned only with positive integer bignums.

We need only one bignum arithmetic operation: multiply a bignum by two to get a new bignum. This operation contains the following sequence of subcomputations; assume the given bignum B is 7 3 6 8:

(1) Multiply the bignum B by 2.
V←B×2≡14 6 12 16

(2) Compute carry for each digit position. Since carries are propagated from right to left, the operator right-propagation (+) accomplishes such a control pattern. A 0 is appended to V to initialize the rightmost carry-in; it also aligns the carry vector C correctly with V.
C←{10≤α+ω} +V,0
V≡ 14 6 12 16
C≡1 0 1 1 0

(3) Add carries to the multiple and normalize each position to a single digit.
W←10|(0,V)+C
(0,V)+C≡1 14 7 13 16
10|(0,V)+C≡1 4 7 3 6

(4) Remove a possible leading zero. Since there is no leading zero in our example, the final answer is just 1 4 7 3 6.
(0=1↑W)↓W

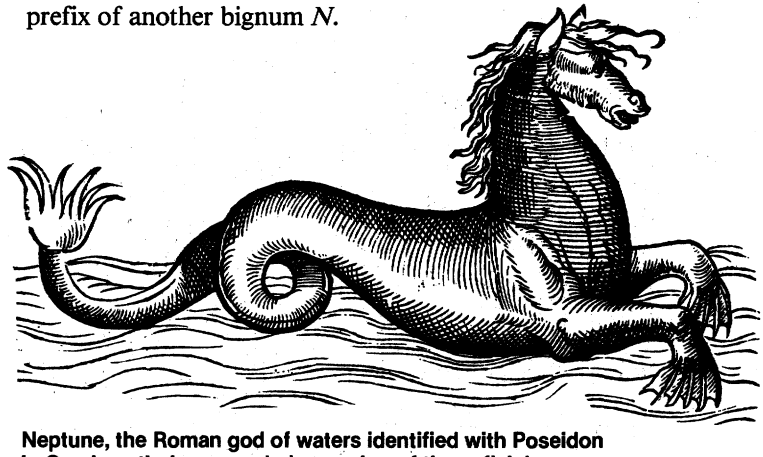
We can put the four expressions together to form a one-liner, as shown in the function double below.

```
DOUBLE B : [1;;1]
  (0=1↑W)↓W←10|(0,V)+{10≤α+ω} +(V←B×2),0
```

By iterating the function double to an initial vector 1, we are able to generate all powers of two as a matrix.

```
⊃DOUBLE  [1]1 ≡ 1
                2
                4
                8
                1 6
                3 2
                6 4
                1 2 8
                . . . .
```

The second part of the solution is to have a Boolean function prefix that determines whether a given bignum Y is a prefix of another bignum N.



Neptune, the Roman god of waters identified with Poseidon in Greek mythology, needed a number of these fish-horse creatures to traverse his domain, which included lakes and rivers as well as seas. Neptune-Poseidon, trident in hand, appears on our cover.

Natural Science Picture Sourcebook, Janet Evens, ed., Van Nostrand Reinhold, 1984

```
Y PREFIX N :[0;1;1] Y = [1]((ρY) ⌊ρN)↑N
```

(If the bignum N is longer than Y, $((\rho Y) \lfloor \rho N)\uparrow N$ truncates N to the same size as Y.)

Applying the function prefix to all powers of two returns a Boolean vector, where the first nonzero of the vector indicates the smallest power of two matched, as shown in the function pptwo.

```
PPTWO Y :[0;;1] (Y PREFIX ⊃DOUBLE⍣[1]2)⍳1
1 2 PREFIX ⊃DOUBLE⍣[1]2 ≡ 0 0 0 0 0 0 0 1 0 0 . . .
PPTWO 1 2 ≡7      Note: 2⁷ = 1 2 8
```

The language described here allows us to take advantage of both functional languages and APL-style programming. It also shows us that infinite carrier arrays combined with their operations (partitions, iter operator, etc.) give us a better programming style than lists and recursion.

We have also shown that, as an expression language, FAC provides tools that allow us to abstract more compact solutions than are possible with other languages. On the other hand, FAC does not address any language issues of large program construction. We deliberately left out these language issues so that FAC could integrate more readily with languages such as Lisp and Pascal. For example, if we extend FAC to have recursions and conditions, they may not improve APL-style programming, but FAC will become a better general-purpose language. □

Acknowledgments

We thank the referees for their valuable suggestions. The project is supported partly by the National Science Foundation under grant MCS-8106181.

References

1. Hai-Chen Tu, *FAC: Functional Array Calculator and Its Application to APL and Functional Programming,* PhD dissertation, Yale University, New Haven, Conn., 1985.
2. Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Comp. Surveys,* Vol. 14, No. 1, Mar. 1982, pp. 93-143.
3. John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM,* Vol. 21, No. 8, Aug. 1978, pp. 613-641.
4. L. Damas and R. Milner, "Principal Type-Schemes for Functional Programs," in *Conf. Record Ninth Ann. ACM Symp. Princ. Programming Languages,* 1982, pp. 207-212.
5. D.B. MacQueen and Ravi Sethi, "A Semantic Model of Types for Applicative Languages," in *Conf. Record 1982 ACM Symp. Lisp and Functional Programming,* 1982, pp. 243-252.
6. *APL2 Language Manual,* IBM Technical Report SB21-3039.
7. Kenneth E. Iverson, "Operators," *ACM Trans. Programming Languages and Syst.,* Vol. 1, No. 2, Oct. 1979, pp. 161-176.
8. Kenneth E. Iverson and Peter K. Wooster, "A Function Definition Operator," *APL Quote Quad,* Vol. 12, No. 1, Sept. 1981, pp. 142-145.
9. Paul Geoffrey Lowney, *Carrier Arrays: An Extension to APL,* PhD dissertation, Yale University, New Haven, Conn., 1983.
10. Eugene E. McDonnell and Jeffrey O. Shallit, "Extending APL to Infinity," in *APL80,* ACM Press, New York, 1980, pp. 123-133.
11. Peter Henderson and John H. Morris, "A Lazy-Evaluator," in *Conf. Record Third Ann. ACM Symp. Princ. Programming Languages,* 1976, pp. 95-103.

Hai-Chen Tu is a member of the technical staff in the Computer Science Laboratory of GTE Laboratories. His research interests include programming languages, software engineering, and semantic databases. He received a PhD from Yale Univeristy, an MS from the University of Iowa, and a BS from National Taiwan University.

His address is GTE Laboratories, 40 Sylvan Rd., Waltham, MA 02254.

Alan J. Perlis is a professor in Yale University's Department of Computer Science. His research interests include programming languages, software engineering, machine architecture, and automatic programming. Previously, he was on the faculties of Purdue University and Carnegie-Mellon University. He has served as editor-in-chief of *Communications of the ACM* and president of the ACM. He received an MS and a PhD from the Massachusetts Institute of Technology, and a BS from Carnegie Institute of Technology (now Carnegie-Mellon University).

His address is Department of Computer Science, Yale University, PO Box 2158, Yale Station, New Haven, CT 06520.