Scaler Functions:
An APL Analysis of the Vertical and Horizontal Implementations

By

Wiley Greiner

B.S. (University of California) 1974

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Engineering

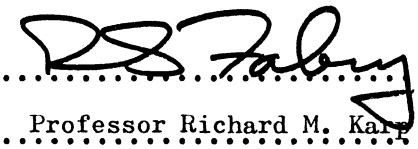
in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved:



..... Robert S. Fabry, Chairman

..... Professor Richard M. Karp (EECS, IEOR)

..... Professor Austin C. Hoggatt (Business Administration)

Committee in Charge

June 12, 1975

SCALAR FUNCTIONS: AN APL ANALYSIS

OF THE VERTICAL AND HORIZONTAL IMPLEMENTATIONS

Wiley Greiner

Master of Science

Electrical Engineering
and Computer Sciences

THESIS ABSTRACT
June 1975

Signature
Thesis Committee Chairman

Plus, Minus, Modulo, Maximum and Nor are examples of scalar functions. If one or more operand is an array, a scalar function performs an identical computation for each element in the result, which is also an array.

Array processors typically support a set of instructions that act like scalar functions. These instructions can be combined to simulate more complex scalar functions, without looping. Alternatively, scalar functions can be simulated as a program which loops once for each element of the array result. These two simulation methods are named the vertical and horizontal simulation methods, respectively.

Both methods have been used on an array processing APL machine designed by the Center for Research in Management Science, where the vertical simulation was found to be much clearer to read and faster, but bulkier, than the horizontal simulation, for most--but not all--scalar functions.

This discovery is one of many conclusions that may apply to similar dynamically allocating array processors, including the next generation of APL emulators.

TABLE OF CONTENTS

Section 1

Introduction

Many useful array manipulations--such as multiplying a vector by a scalar to obtain a vector, and adding together two matrices of the same shape to obtain a third--are called scalar primitives. The APL computer language provides a large collection of built-in scalar primitives. All scalar primitives have one result, and at least one source operand. The I^{th} element of the result is computed using an element from each source operand. If a source operand contains exactly one element, then that element is used. Otherwise, the I^{th} element is used. Comprehensive definitions will be presented when more key terms have been described.

The APL@CRMS[1] host machine[2] executes a set of instructions, including Times and Plus, that act like scalar primitives. These instructions are known as scalar instructions. Other instructions, such as branch and index, do not act like scalar primitives. The Scalar Instruction Set contains all scalar primitives that correspond to exactly one machine instruction. The remaining scalar primitives are

---

[1] P. McJones, C. Grant & W. Greiner, CRMS APL Processor Manual, Center for Research in Management Science, University of California, Berkeley (Revised May 30, 1974).

[2] The APL@CRMS host machine is microprogrammed to emulate a subset of the APL language. For more information about this machine, consult the Digital Scientific Corporation META 4 Computer System Microprogramming Reference Manual, publication number 7043MO (March 1972).

bootstrapped from more than one instruction. This paper examines the two bootstrap methods used on the APL@CRMS system: the horizontal and the vertical methods. The horizontal method computes each element of the final result (from start to finish) before beginning to compute the next element of the final result. A program that adds a vector to a scalar then multiplies the sum by another scalar, by looping once for each element in the vector, is an instance of the horizontal method. The vertical method performs a single-instruction operation on all intermediate results before executing the next single-instruction operation. Adding a vector to a scalar then multiplying this sum by another scalar, using the Plus and Times scalar instructions, is an instance of the vertical method. Here, the intermediate result--a vector of sums--has a shape equal to the shape of the final result. In general though, intermediate results may have more, or less, elements than the final result.

One or more scalar instructions occur in each APL@CRMS vertical implementation. These implementations take advantage of the array processing features of scalar instructions. Unfortunately, the existing vertical implementations are not yet understood well enough to be systematically adapted for any arbitrary scalar primitive. Or, more to the point, discovering a vertical implementation is a heuristic problem. Yet many aspects of the vertical, array processing method are beginning to be understood. The vertical method, because it

lacks conditional branching, should be of particular interest to users of parallel and non-backup pipelined machines. This paper compares interesting aspects of the vertical method to similar aspects of a universally-adaptable, looping, horizontal method. Dynamic storage allocation and deallocation demands, relative execution speeds, program style and clarity, user error detection, time- and space-efficiency, and both theoretical and practical limitations are among the aspects compared.

The APL community has long been aware of the array processing features of one-line programs that use scalar primitives. A 1971 APL reference manual,[3] for example, spoke of "programs that were originally expected to work on single numbers, but which turn out to work just as well on vectors of numbers. . .It isn't clear how this can be done." Although many APL references notice these powerful one-line programs, few attempt to analyze the tradeoffs between non-looping programs and looping programs that do the same thing. This paper attempts one such analysis.

---

[3] APL/360 Primer, IBM number GH20-0689-2, p. 110 (1971). This primer is a second revision. Interestingly enough, neither the original nor the first revision contain a similar quotation.

Definitions and Formalizations

SCALAR FUNCTIONS

Scalar functions have one, two or more source operands, and return exactly one result. This result is scalar only if all source operands are scalars. The result is an array if any source operand is an array. The shape[4] of the result of a monadic[5] scalar function is the same as the shape of the lone source operand. R, the shape of the result of a non-monadic scalar function, depends on $\{S_i: 1 \leq i \leq N\}$, the shapes of the N source operands. The four possibilities are:

a) If the set $S' = \{S_i: S_i \neq \text{singleton-shape}\}$,[6] contains one or more shape, all of which are identical, then R is the same as the shapes in $\{S_i\}$ that are not singleton-shapes.

b) Otherwise, if S' is the empty set, then R is the same as the longest of the S_i vectors.

c) Otherwise, if the lengths of any two vector members

---

[4]The shape of an APL variable is a vector of dimensions. The shape of a 2 by 3 by 4 array, for example, is 2 3 4. The shape of a scalar is the empty vector.

[5]Monadic means one source operand. Dyadic means two source operands.

[6]Singleton-shapes are vectors that contain only numeric ones. Singletons are single element variables, such as scalars, single element vectors, and 1 by 1 matrices.

of S' are unequal, then generate the "RANK ERROR"[7] trap. No result is returned.

d) Otherwise, some elements in some vector members of S' are unequal. Generate the "LENGTH ERROR" trap. No result is returned.

In summary:

R, the shape of the result:--

--depends on the shapes in $\{S_i\}$, and not on the values of the elements in the source operands.

--is the same as one or more shapes in $\{S_i\}$.

--would be unchanged if the ordering of the operands were rearranged.

Scalar functions:--

--are never functions of less than one variable.

--give the illusion that identical logic is re-executed once for each element of the result. (The actual implementation is not required to follow the illusion.)

Here is the Fundamental Criterion of Scalar Functions:--

--The result of a scalar function never depends on the order in which the elements of the result are computed.

In fact, the elements might as well be computed in parallel. The Criterion will be invoked later to distinguish scalar functions from the similar monadic "?" function (Roll). The Roll function generates pseudo-random positive integers,

---

[7]The rank of an APL variable is the number of dimensions it has. Or, more precisely, an APL variable's rank is the shape of its shape. The ranks of a scalar, vector, matrix, and N-dimensional array are 0, 1, 2, and N, respectively.

analogous to the "roll of the dice."

Scalar primitives are defined as scalar functions that happen to be APL language primitives. All scalar primitives are either monadic or dyadic scalar functions. A Non-Instruction Scalar Primitive is any scalar primitive which is not functionally equivalent to one APL@CRMS instruction. Non-Instruction Scalar Primitives are simulated with more than one APL@CRMS instruction. These simulations must be careful to generate the correct error trap whenever operands are outside their domains. Currently, two types of scalar function simulations have been implemented on the APL@CRMS system: the horizontal Iterating Function and the vertical Scalar Expression simulations. They are described below.

ITERATING FUNCTIONS: THE HORIZONTAL IMPLEMENTATION

The looping Iterating Function (IF) method re-executes some logic once for each element of the result.

Definition:--

--An Element Iteration is the horizontal logic which is computed once for each element of the result by an Iterating Function. The result must be independent of the order of iteration. This latter requirement stems from the previously defined Fundamental Criterion of Scalar Functions.

The typical IF contains three parts: the front-end part, the re-executing Element Iteration part, and the back-end part. A front-end part for a monadic scalar function might look like:[8]

```
]  S:=.SHAPE.X     &  S is the shape of the operand, X.
]  X:=.RAVEL.X     &  Ravel X into a vector.
]  I:=.SHAPE.X     &  I is the length of this vector.
]  R:=X            &  Initialize the result, R.
```

The re-executing Element Iteration part of the Signum--or "sign of" function--might look like:

```
]  START:   GOTO END IF I=0
]          R[I]:=1              & Assume positive.
]          GOTO DONE IF X[I]>0
]          R[I]:=0              & Assume zero.
]          GOTO DONE IF X[I]=0
]          R[I]:=-1             & Necessarily negative.
]  DONE:    I:=I-1
]          GOTO START
]  END:
```

And the back-end part might look like:

```
]          R:=S.RESHAPE.R
```

Notice how the above IF performs a special test designed to determine whether the source operand is an empty[9] array. Also notice that the re-executing Element Iteration part

---

[8]Raw APL lines are denoted by a right bracket, "]", on the left margin. If an APL line contains an "&", then everything to the right of the leftmost "&" is comment.

[9]An array is <u>empty</u> if it contains no elements. Length zero vectors and 2 by 0 matrices are examples of empty vectors and matrices, respectively.

indexes into a ravelled[10] version of the operand. The operand had to be ravelled into a vector so that each element could later be indexed, one at a time. The expense of front-end ravelling and back-end reshaping could have been avoided if APL@CRMS contained an instruction that indexed the I^{th} element of any APL variable, independent of its dimensionality. Adding such an instruction to the APL@CRMS non-scalar instruction set could improve IF time and space efficiencies.

SCALAR EXPRESSIONS: THE VERTICAL IMPLEMENTATION

A Scalar Expression, unlike an Iterating Function, does not loop. In other words, a Scalar Expression is executed once for each result, rather than once for each element of the result. Scalar Expressions compute multiple element results in just one pass due to the array processing nature of each scalar instruction. Scalar instructions may internally[11] invoke logic that is sequential, parallel, pipelined, or even magical in nature. The internal logic is transparent to the software programmer. Certain hardware designs may run faster

---

[10]The ravel of any N element APL variable returns an N element APL vector. The N elements are stored in row-major order (like PL/I, and unlike FORTRAN). Ravelling a scalar, a 2 by 3 matrix, and a vector returns a single element vector, a six element vector, and an unchanged vector, respectively.

[11]APL@CRMS scalar instructions invoke very fast microcoded loops.

than others, but for our purposes, it does not matter that, say, a particular sequential machine runs faster or slower than another parallel machine.

Definitions:--

--A S Scalar Expression, where S is a set of scalar functions, is an APL expression that meets the following syntactic qualifications:

1. All of its function calls are to functions in the set S, and

2. All of its constants are scalars.

--A scalar-bound variable is any variable that is known to be a scalar: its rank is zero.

--A Monadic(Dyadic) S Scalar Expression is a S Scalar Expression that contains exactly one(two) variable(s) that is(are) not scalar-bound.

If one understands the above definitions, the following unproved assertions should be plausible:--

--A Monadic S Scalar Expression returns a monadic scalar function of its only variable that is not scalar-bound.

--A Dyadic S Scalar Expression returns a dyadic scalar function of its two variables that are not scalar-bound.

We will frequently refer to an APL@CRMS Scalar Instruction Set Scalar Expression. These seven words will be abbreviated as SE.

DUALITY AND FUNCTIONAL EQUIVALENCE

Many languages besides APL support scalar functions. APL, however, is one of the few languages that would evaluate an expression like

A:=(B+C)xD

vertically. If B, C and D are equally shaped arrays, then APL must allocate storage for the intermediate result array, B+C. Most other languages that support scalar functions, such as ALGOL, PL/I, and some BASICs would evaluate the above expression horizontally, as one implicit do-loop. In other words, each element A[i] of the final result is computed, from start to finish, before beginning to compute the next element.

Definitions:--

--The dual of a SE is an IF that executes the SE horizontally as an Element Iteration: The SE is evaluated, with scalar arguments, once for each element of the result.

--The dual of the dual of a SE, is the SE itself.

Every SE has a dual. But not every IF has a dual, because an IF's Element Iteration need not conform to the constraints[12]

---

[12]Refer back to the definition of an S Scalar Expression, given earlier in this section.

that Scalar Expressions must conform to. If an Element Iteration contains any conditional branching, then the Element Iteration's IF has no dual.

A SE and its dual are functionally equivalent. Two algorithms are said to be functionally equivalent if, whenever they both compute results, the two results are equivalent. However, when both generate errors in lieu of results, these errors need not be equivalent. This is because multiple user errors may be detected in a different order, by the two algorithms. Assume, for example, the existance of two functionally equivalent simulations of the Divide scalar primitive. The two might generate different errors for the expression

```
] (1E¯30 6 9) ÷ (1E34 2 0) & which performs a scalar
```

division between two three-element vectors. Here, three divisions must be computed, two of which contain user errors. One simulation might try to compute 1E¯30 ÷ 1E34, and detect an underflow error. A functionally equivalent simulation might try to compute 9 ÷ 0 first, instead, and detect a divide by zero error.

Also, if an algorithm successfully computes a result, a functionally equivalent algorithm might generate an error. A SE may successfully compute a result, for example, while its functionally equivalent dual generates the "NO MORE SPACE LEFT" trap.

APL@CRMS Standards

APL@CRMS supports the following scalar primitives, some of which are Non-Instruction Scalar Primitives.

Monadic:--

No_Change	
Change_Sign	
Signum	(sign of)
Reciprocal	(1 divided by)
Exponential	(e raised to)
Logarithm	(natural log of)
Ceiling	(least int not <)
Floor	(greatest int not >)
Magnitude	(absolute value of)
Factorial	(gamma function of)
Pi_Times	
Not	
ID	(internal type of)
Translate	(convert between types)

Dyadic:--

Plus	
Minus	
Times	
Divide	
Power	(raise to a power)
Logarithm	(log to a base)
Maximum	
Minimum	
Residue	(similar to remainder)
Combinations	(complete beta function)
Circle	(sin, cos, tan,
(	arcsin, arccos, arctan,
(	sinh, cosh, tanh,
(	arcsinh, arccosh. . .)
And	
Or	
Nand	
Nor	
Less_Than	
Not_Greater	
Equal	
Not_Less	
Greater	
Not_Equal	

An early design goal of APL@CRMS was to be similar to APL\360. APL\360 at the time was the dominant APL implementation. So, it was felt APL@CRMS programmers would probably have had prior APL\360 experience.

There are many differences[13] between the APL\360 and the APL@CRMS implementations, however. The two were designed for different purposes: APL@CRMS was designed primarily to run extremely interactive management science experiments; APL\360 wasn't. APL@CRMS is almost a decade newer. It incorporates newer hardware technologies and software strategies. Yet, most of the differences appear so well thought out that studies, such as this, performed on APL@CRMS should apply to existing APL implementations, as well as future implementations now on the drawing board.

A few of the differences affect scalar functions, indirectly. One such difference is a new data-type. Mixed arrays, which existed in the earliest description of APL,[14] are arrays that contain both numbers and characters. APL\360 does not support mixed arrays, whereas APL@CRMS does. Special scalar primitives exist in APL@CRMS for manipulating mixed arrays. These primitives comprise the set of mixed scalar primitives. This set never existed on APL\360. APL@CRMS was

---

[13] W. Greiner, APL@CRMS Users Guide, Center for Research in Management Science, University of California, Berkeley (To Be Published).

[14] K. E. Iverson, A Programming Language (1962).

required to establish standards for the handling of mixed arrays, through trial and error. It now appears that the set of mixed scalar primitives, as currently implemented is incomplete. The issue of completeness will be discussed further in the subsection, "A SCALAR INSTRUCTION SET THAT IS NOT COMPLETE."

Expressions which contain more than one assignment symbol (":=") are called pornographic. Some APL implementations evaluate some pornographic expressions in a hard-to-anticipate manner. APL@CRMS, however, always evaluates these expressions according to a straightforward, predictable rule: all multiple assignments are evaluated in strict, right-to-left, order. Thus, the expression

```
]     ((A:=2)+A)+A:=1    & for example,
```

returns 4, in APL@CRMS. The same expression might return 5 or 6 on some IBM-based APL implementations, such as .APL.SV., APL*PLUS, and PCS\APL. The subsection, "MULTIPLE ASSIGNMENTS," discusses some reasons for using expressions that contain more than one assignment symbol.

The APL@CRMS Scalar Instruction Set:

```
] .ID.Y        &  (ID) Returns 1 if Y is numeric, 0 if character.

] .TRANS.Y     &  (Translate) Converts from character to integer,
]              &  and integer to character.  The "DOMAIN ERROR"
]              &  is generated if Y is numeric, but not an
]              &  integer between 0 and 255.

] X=Y          &  (Equals) Returns 1(true) or 0(false).  See below
]              &  for traps that may occur when X and Y are numeric.

] &  All following instructions will generate the "TYPE ERROR"
] &  if any element of their operand(s) is non-numeric.

] +Y           &  Returns Y.

] -Y           &  (Change_Sign) Returns Y subtracted from 0.

] .FLOOR.Y     &  Returns the greatest integer not greater than Y.

] .CEILING.Y   &  Returns the least integer not less than Y.

] |Y           &  (Magnitude) Returns the greater of Y and -Y.

] X+Y          &  (Plus) Dyadic + - x * < and = may generate the

] X-Y          &  (Minus) "FLOATING POINT OVERFLOW ERROR," or the

] XxY          &  (Times) "FLOATING POINT UNDERFLOW ERROR" traps.

] X*Y          &  (Divide) May generate the "DIVIDE BY ZERO ERROR."

] X<Y          &  (Less_Than)

] &  All following instructions will generate the "DOMAIN ERROR"
] &  if any element of any operand is non-binary.

] ~Y           &  (Not)

] X.AND.Y      &  (And)

] X.OR.Y       &  (Or)
```

A goal of the APL@CRMS Scalar Instruction Set is to detect user errors very soon after they occur, so as to minimize error propogation.

Some Nitty-Gritty Vertical Examples

EXAMPLES OF THE VERTICAL STYLE

Vertical programs do not look like horizontal programs. Their styles are different. Good vertical programs often seem hideously inefficient and sloppy to someone with only horizontal experience. Since vertical programs are not yet commonplace, this section will go over a few SE examples. These examples will be drawn from scalar primitives, whenever possible.

A few Non-Instruction Scalar Primitives have very obvious SE's:

```
] ÷Y          & (Reciprocal) is the 1.0÷Y SE.
] X≠Y         & (Not_Equal) is the -X=Y SE.[15]
] X>Y         & (Greater) is the Y<X SE.
] X≤Y         & (Not_Greater) is the (X<Y).OR.X=Y SE.
] oY          & (Pi_Times) is the 3.14159265xY SE.
```

Most Non-Instruction Scalar Primitives, however, have somewhat less obvious SE's. The dyadic "|" scalar primitive (Residue), for instance, is defined as follows:[16]

---

[15] APL expressions are always evaluated right-to-left, except that parentheses behave in the usual way. Aside from this rule, there are no operator precedence rules, whatsoever.

[16] A|B in APL is similar to B modulo A, the remainder of B divided by A.

```
] A|B          & for A≠0,          is B-(|A)x.FLOOR.B÷|A
]              & for A=0 and B≥0, is B
]              & for A=0 and B<0, generates the "DIVIDE BY
]              &                  ZERO ERROR" trap.
```

In APL@CRMS, Residue is implemented as one SE which covers all three cases:

```
] X|Y          & is the Y-(|X)x.FLOOR.Y÷|X+(Y<0)<X=0 SE.
```

This tricky code is necessary because vertical programs may not contain conditional branches.

EXAMPLES OF INSTRUCTION IDIOSYNCRACIES

The monadic "x" scalar primitive (Signum) also has three cases. If the source operand is positive, negative, or zero, monadic "x" returns 1, ¯1, or 0, respectively.

```
] xY           & (Signum) is the (0<Y)-Y<0 SE.
```

The Signum SE is well-defined for all values of Y largely because of this very important characteristic of the + - x ÷ < and = scalar instructions: None will give inaccurate results, or generate the unexpected "FLOATING POINT OVERFLOW ERROR" or "FLOATING POINT UNDERFLOW ERROR" traps if either source operand is zero. Unfortunately, there can be messy roundoff and other problems if both operands are nonzero:

```
] X.MAX.Y      & (Maximum) is the X+(Y-X)xX<Y SE, or is it??
```

The answer is "no," since:--

--for a very negative X and a small Y, say X=-1E30 and Y=2, the subexpression (Y-X) returns -X, due to a common roundoff idiosyncracy. And, at best, zero is only "close to" (-1E30).MAX.2.

--for a very negative X and a very positive Y, and vice-versa, the subexpression (Y-X) generates the "FLOATING POINT OVERFLOW ERROR" trap, which is never an appropriate trap for X.MAX.Y.

A better simulation of

```
] X.MAX.Y      & is the (YxX<Y)+Xx-X<Y SE.
```

Sometimes even this SE generates an unexpected error. When X and Y are sufficiently close together, but not equal, X<Y generates the "FLOATING POINT UNDERFLOW ERROR" trap. X<Y in microcode computes X-Y, which can cause underflow problems because APL@CRMS does not support unnormalized floating point numbers. The problem, as numerical analysts know, lies with the "<" scalar instruction. However, any microprogram patch that corrects the "<" problem implicitly fixes the ".MAX." problem, too. This example suggests a prudent precaution: Since the characteristics of one scalar instruction may have a strong influence upon the characteristics of many SE's, all hard-to-anticipate idiosyncracies should be removed from

scalar instructions. (Many hard-to-anticipate arithmetic idiosyncracies in APL@CRMS could easily have been eliminated early in the design by heeding the advice of a competent numerical analyst.)

These idiosyncracies tend to affect vertical programs more than horizontal programs. This is because the latter may conditionally branch around special case code whenever an operand is out of bounds. Thus, in the latter, the programmer need only test his code between narrow programmer-defined bounds. Special case vertical code, on the other hand, could conceivably be required to process any operand value, whatsoever.

EXAMPLES OF LOCALIZATION

If both X and Y were singletons, the double evaluation of X<Y by the above Maximum SE would be trivial. But, since X or Y may be monstrous arrays, a viable alternative for

```
] X.MAX.Y      & is the (YxTEMP)+Xx-TEMP:=X<Y SE,
]              & followed by FREE(TEMP).
```

SE's defined from now on may contain the Maximum primitive even though Maximum does not belong to the APL@CRMS Scalar Instruction Set. This is because the notation, ".MAX.", may be considered an <u>abbreviation</u> of the Maximum SE. The abbreviation principle applies to all SE simulations, and not just the Maximum SE.

EXAMPLES OF NON-DISTRIBUTION

Many mathematical Non-Instruction Scalar Primitives evaluate a fixed number of terms in one of many selectable series. The series selected depends on the source operand, X. Series 1, for example, would be used if 0<X<1; while series 2 would be used if 1≤X<3.14159; and so-on. This logic could be implemented as the SE:

(SER1x(0<X).AND.X<1) + (SER2x(1≤X).AND.X<3.14159) + ...

Most of the time, the first few operations of SER1, SER2 , etc., are identical. Then, too, a few series may share operations that are not shared by all the series. The better-written mathematical SE's tend not to distribute multiplication over addition. A non-distributive SE might look like:

```
SER_START + (SER_MID1+(SER_END11x...) + SER_END12x...)
   + (SER_MID2+(SER_END21x...) + SER_END22x...)
   + ...
```

Not distributing multiplication over addition increases SE time-efficiency by reducing the number of scalar instructions executed. More will be said about the strong relationship between SE syntax and SE efficiency in the "SE SPACE-EFFICIENCY" and the "TIME-EFFICIENCY" subsections.

EXAMPLES OF NON-SIMULATION APPLICATIONS

SE's have many applications in additon to simulating Non-Instruction Scalar Primitives. Recall that all SE's return scalar functions.

The "convert small letter into capital letter" monadic scalar function could be simulated as the following SE:

```
]  & P is the operand of the "convert small letter
]  & into capital letter" scalar function.
]  & Integer SMALL is the scalar constant .TRANS.'a' .
]  & Integer DIFF is the constant (.TRANS.'A')-.TRANS.'a' .
]  .TRANS. CP + DIFFx(SMALL≤CP).AND.(CP:=.TRANS.P)<26+SMALL
```

This SE assumes that the character codes for small 'a' through small 'z' are contiguous, as are the codes for capital 'A' through capital 'Z'. The next "convert small letter into capital letter" SE makes no such continuity assumptions:

```
.TRANS. CP + (((.TRANS.'A')-.TRANS.'a') x CP=.TRANS.'a')
       + (((.TRANS.'B')-.TRANS.'b') x CP=.TRANS.'b')
       + (((.TRANS.'C')-.TRANS.'c') x CP=.TRANS.'c') . . .
       . .
       . .
       +((.TRANS.'Z')-.TRANS.'z')x(CP:=.TRANS.P)=.TRANS.'z'
```

SE's that contain a large amount of conditional logic tend to be long. But fortunately these SE's can be very easy to program, as the next sections will show.

Two Advantages of Programming Vertically

CLARITY

Most people prefer to program horizontally. Perhaps vertical programs seem too complicated, since vertical instructions sometimes process large arrays, whereas horizontal instructions work with simple scalars, only. And, many people believe that array programming is less natural for human beings than sequential programming. This belief cannot always withstand scrutiny. Let's look at the problem of adding two equally shaped vectors, X and Y, placing the vector of sums in Z. Here is one horizontal approach, using FORTRAN:

```
9     DO 10 I=1,N
10    Z(I)=X(I)+Y(I)
```

I doubt whether DO-LOOP syntax looks natural to the novice. Furthermore, the above DO-LOOP solves only part of the problem. N, the length of X, is not given. Z must be statically dimensioned for the worst case, since FORTRAN does not support a dynamic storage allocator. A better example of the horizontal approach is the following APL program, which implicitly uses APL's dynamic storage allocator:

```
]         Z:=X              & Z and X have the same shape.
]         I:=.SHAPE.X       & I becomes the shape of X.
]  LOOP: EXIT IF I=0
]         Z[I]:=X[I]+Y[I]
]         I:=I-1
]         GOTO LOOP
```

Now, contrast the naturalness of the above example with that of the vertical approach using APL:

```
]       Z:=X+Y
```

The addition of vectors problem is one of many scalar functions that can be stated more clearly vertically than horizontally. Horizontal programs, at least in APL, are notoriously unreadable; whereas vertical programs can be extremely clear and concise.

GENERALITY

Scalar Expressions offer many programming conveniences not found in conventional, horizontal programs. Since Scalar Expressions are unaware of the shapes of their operands, the Scalar Expression programmer is not responsible for ravelling or reshaping arrays, or looping, or incrementing and testing nested counters. Scalar Expressions are never written to work on an operand with a specific shape: instead, they are always written to handle the more general case of any operand shape, whatsoever. And, Scalar Expressions have a surprisingly wide range of applications. In fact, virtually all the APL@CRMS scalar primitives have already been implemented as SE's.

Two Useful Vertical Programming Tools

Unfortunately, SE programming usually takes considerable thought. I hope that Scalar Expression programming eventually will become more standard, and less of an art. With this goal in mind, let me list two Scalar Expression programming techniques.

MULTIPLYING BY A BINARY VALUE

Conditional branching can be approximated by summing up a series of products, each consisting of a term multiplied by the binary (1 or 0) result of a relational subexpression. Multiplying a term by a binary value yields either the term itself or zero. This technique makes the monadic "x" SE very trivial to derive.

```
]  xY     & is (1x0<Y) + (¯1xY<0) + 0xY=0
]         & which can be simplified into (0<Y)-Y<0
```

It is easy to conditionally induce errors, as in

```
]  EXPR + (-1+B<0) + -1+3.1415926<B
```

which generates the "DOMAIN ERROR" trap unless every element in B is between zero and Pi. (The domain of the "-" primitive is the set {0,1}).

The disadvantages of this technique stem from the fact that certain intermediate results--those with no bearing on the final result--are computed first, then multiplied by zero. Two of the disadvantages are (1) inefficiency caused by performing needless computation, and (2) hard-to-anticipate

errors which occur because no portion of a SE may be conditionally bypassed when operands are out of bounds.

MULTIPLE ASSIGNMENTS

Multiple statements can be approximated with multiple assignments. Although multiple assignments are never necessary, they can significantly reduce SE complexity. Take, for example, the problem of raising X to the non-negative integral power P, by performing up to Pmax successive multiplications. Normally, the SE complexity grows by Order(Pmax). But, through multiple assignments, the SE complexity grows by only Order(log Pmax). (For clarity, the multiple assignment SE is represented as a sequence of smaller SE's, one per line):

```
]  L:=(LxY:=X) + ~L:=(I+I:=.FLOOR.P÷2)<P
]  & previous line handles Pmax=2-1=1
]
]  L:=((LxY:=YxY) + ~L:=(I+I:=.FLOOR.I÷2)<I))xL
]  & previous line handles Pmax=4-1=3
]
]  & next line handles Pmax=8-1=7
]  L:=((LxY:=YxY) + ~L:=(I+I:=.FLOOR.I÷2)<I))xL
```

Adding on N more lines exactly like the last pushes Pmax to $2^{N+3}-1$.

Unfortunately, all multiple assignment SE's are pornographic, and so may evaluate unpredictably on certain IBM-based APL implementations.

The APL@CRMS SE Implementation

Almost all Non-Instruction Scalar Primitives have been implemented, in APL@CRMS, as SE's. The smaller SE primitives are installed as in-line macros, which are "called" immediately after their operands are pushed onto the stack. There are three temporary variables, which are global and are provided for the exclusive use of in-line macros. These variables are not accessable to the problem program. They are, however, available to the debugging programmer. In-line macros which assign values to temporary variables must explicitly FREE these variables, before "returning." This makes temporary variables appear to be local. In practice, temporary variables are used to memorize operands before they have been popped from the stack. The monadic "x" macro (Signum or (0<Y)-Y<0), for example, is:

```
(a)   ASSIGN TEMP1        (copy top of stack to TEMP1)
(b)   PUSH numeric 0      (onto top of stack)
(c)   INTERCHANGE         (top 2 slots on stack)
(d)   LESS_THAN           (replace top 2 slots with <)
(e)   PUSH TEMP1          (onto top of stack)
(f)   PUSH numeric 0
(g)   LESS_THAN
(h)   SUBTRACT            (result is now on top of stack)
(i)   FREE TEMP1          (localize TEMP1)
```

The larger SE primitives are installed as calls to APL@CRMS RUNTIME functions. The Call non-scalar instruction instructs the microprogram to add a new frame on the stack, update a few state words, and initialize any local variables. The Return instruction reverses the process, and implicitly de-allocates all local variables.

Compared with the RUNTIME installation of SE primitives, the macro installation is

--faster because there are no Call or Return instructions executed.

--bulkier because the macros usually expand into more code than that of a single Call instruction.

Reducing the Scalar Instruction Set

The APL@CRMS Scalar Instruction Set can be reduced from what is currently supported without losing any functional properties.

Definition:--

> --A set SS of scalar functions is a Reduced Set if all functions in the APL@CRMS Scalar Instruction Set can be simulated by SS Scalar Expressions.

The APL@CRMS Scalar Instruction Set is a Reduced Set, by definition. A more interesting Reduced Set, SS, is

SS = {.TRANS., =, .CEILING., dyadic -, x, +, <, -}

Notice that every function in SS also appears in the APL@CRMS Scalar Instruction Set, although this is not a prerequisite for SS to be a Reduced Set. We must show that all functions in the APL CRMS Instruction Set, but not in SS, can be simulated by SS Scalar Expressions.

```
] .ID.Y        & is the 1-(Y='a')-0-(Y='b')...-(Y='Z') SE.[17]
]              & (There are 256 APL@CRMS character codes.)

] +Y           & is the 0-0-Y SE.

] -Y           & is the 0-Y SE.

] .FLOOR.Y     & is the 0-.CEILING.0-Y SE.
```

---

[17] All SS Scalar Expressions are also SE's, since SS is a subset of the APL@CRMS Scalar Instruction Set.

```
] |Y          & (Magnitude) is the Yx1-2xY<0 SE.

] X+Y          & is the X-0-Y SE.

] X.OR.Y       is the -(--X)<-Y SE.  The double negation
]              & generates "DOMAIN ERROR," when appropriate.

] X.AND.Y      & is the (-X)<--Y SE.
```

If APL@CRMS numbers were represented in two's complement form,[18] then the 0-0-Y SE would not correctly simulate monadic "+" for all values of Y.

There may be SE's which simulate Times, without using the "x" scalar instruction. Due to the finite precision of "x" and "+", the X+1+Y SE does not correctly simulate "x" for all values of X and Y. One case in point is Y=0.

The "+" primitive cannot be simulated by SS-{"+"} Scalar Expressions because "+" is the only function in SS capable of generating the "DIVIDE BY ZERO ERROR" trap. But, the "-" primitive can easily be simulated by a SS-{"-"} Scalar Expression. Translate can generate the "DOMAIN ERROR" trap, which is a necessary part of the Not simulation.

SS would still be a Reduced Set if certain pairs of functions were replaced by single, new functions. But the above examples should serve to caution Scalar Expression programmers to pay very careful attention to the

---

[18]The set of two's complement numbers is not closed under the Change_Sign function. This is because a two's complement number, N, is equal to 1+BITWISE_COMPLEMENT_OF(-N).

idiosyncracies of both the simulated and the simulating functions.

```
]             & Replace = and < with ≤.  Now, define
]  X=Y       & as the (X≤Y)xY≤X SS Scalar Expression, and
]  X<Y       & as the (X≤Y)x-Y≤X SS Scalar Expression.
```

Unfortunately both definitions are incorrect when X and/or Y contain characters. To be more precise, the "=" and "<" SS Scalar Expressions violate two rules:--

--that X=Y must return 0 when X or Y, but not both, is character, and

--that X<Y must generate the "TYPE ERROR" trap when X and/or Y is character.

A more suitable building block than "≤" is "≠", where

```
]  X≠Y       & returns 1 if X=Y.  Otherwise, it
]             & returns 0 if X and/or Y is character and X≠Y.
]             & If neither applies, X<Y is returned.  Now,
]  X=Y       & becomes the (X≠Y)xY≠X SS Scalar Expression.
]  X<Y       & becomes (X≠Y)x-Y≠X-Y-Y SS Scalar Expression.
```

The extra subtractions generate the necessary "TYPE ERROR" when one or both operands of "<" is character.

The following transformation of a Reduced Set yields another Reduced Set: Replace all monadic functions with one dyadic function that dispatches on its left operand.

Three Scalar Expression Limitations

A SCALAR INSTRUCTION SET THAT IS INCOMPLETE

A small number of functions that can be simulated by Iterating Functions apparently cannot be simulated by any APL@CRMS Scalar Instruction Set Scalar Expression. This appears to be caused by minor oversights in the selection of the APL@CRMS "base set" of scalar instructions. One example appears to be the "translate right operand if the left operand is zero(false), else pass right operand unchanged" dyadic scalar function. This function, which will be called Translate2 and be denoted as ".TRANS2.", could be added to both the APL@CRMS Scalar Instruction Set and to the set of mixed scalar primitives, and perhaps should be. The addition would make it possible to implement many functions as SE's which cannot now be implemented as SE's. One such function is discussed in the next paragraph.

As was earlier mentioned, APL 360 does not permit arrays to contain both numbers and characters. APL@CRMS on the other hand, does permit these mixed arrays. Unfortunately the APL@CRMS Scalar Instruction Set was developed without sufficiently considering what scalar functions should be provided to act on mixed arrays. All newly considered scalar functions must therefore be implemented as SE's or IF's. Usually, the IF implementation is almost trivial; whereas the SE implementation is next to impossible.

Consider the dyadic scalar function called Add2, defined as follows: The left operand, L, must be numeric; the right operand, R, may be numeric or character; RESULT is the sum of L and R if R is numeric, otherwise RESULT is the character in R.

```
] 1 .ADD2. 'I=' 5    & for example,
```

would return the three element mixed vector, 'I='6. And,

```
] (0 0 1 17.3 2) .ADD2. '2x' 3 '=' 6
```

would return the five element mixed vector, '2x'4'='8. Add2 is easy to implement as an IF. The Element Iteration might look like:

```
]     RESULT:=L
]     EXIT IF ~.ID.L     & Exit if character.
]     RESULT:=L+R        & Else compute sum.
```

Because of the conditional exit, this IF has no dual. In fact, I doubt that Add2 can be implemented as any SE. Add2 can be implemented, however, as the following Scalar Instruction Set union Translate2 Scalar Expression:

```
]     T.TRANS2. (TxL) + (T:=.ID.R).TRANS2.R
```

Add2 is one of many scalar functions which probably should be made available to act on mixed arrays. Other similar functions are Minus2, Times2, Power2, Logarithm2, and

Maximum2. None of these appear to be implementable as SE's. All can be implemented as IF's, trivially. And, all could be implemented as SE's if the Scalar Instruction Set included Translate2.

A FUNCTION THAT IS NOT QUITE SCALAR

Consider the monadic "?" primitive (Roll):

```
] R:=?Y          & assigns to R the Roll of Y.
```

Each element in Y must be a positive integer. R has the same shape as Y, and each element in R is a pseudo-random integer between one and the corresponding element in Y. For example,

```
] ? 10 10 2      & might return the vector 6 1 2.
```

To the casual observer, monadic "?" looks like a scalar function. But, easily unnoticed by the casual observer is what will be called the "rule of repeatable pseudo-random number generators." Namely, the next usage of the generator accesses a so-called seed, which was set during the last usage. Thus, the elements in the result of monadic "?" cannot be computed in parallel, but must be computed one-by-one, and in some proper sequence.[19]

Monadic "?" cannot be simulated by any Scalar Expression. A formal justification of this statement must

---

[19] In APL@CRMS, the monadic "?" sequence is right-to-left. The expression, "?3 4", for example, first processes the 4 (and sets the seed), then accesses the seed while processing the 3. Thus, for any vectors A and B, "?A,B" is functionally identical to "(?A),?B".

wait until after the concepts of multiple- and single-instruction Element Iterations are introduced.

```
C     FORTRAN-IZED EXAMPLE OF AN ELEMENT ITERATION.

      DO 10 I ∈ 1,N
      TI=C(I)-D(I)
10    A(I)=B(I)+TI
```

The "I ∈ 1,N" notation means the programmer is convinced the result, A, is independent of the order of iteration.

An Element Iteration, like the above example, only with more than one APL@CRMS scalar instruction per loop is called a multiple-instruction Element Iteration. Element Iterations with exactly one such instruction per loop are called single-instruction Element Iterations.

Assertions:--

--Each function in the APL@CRMS Scalar Instruction Set performs a single-instruction Element Iteration, internally.

--All Scalar Expressions are functionally equivalent to a permutation of single-instruction Element Iterations. And, all permutations of single-instruction Element Iterations are functionally equivalent to a Scalar Expression.

```
C     FORTRAN-IZED EXAMPLE OF A PERMUTATION
C     OF SINGLE-INSTRUCTION ELEMENT ITERATIONS.

      DO 20 I e 1,N
20    T(I)=C(I)-D(I)

      DO 30 I e 1,N
30    A(I)=B(I)+T(I)
```

Notice the temporary, T. This large vector was missing from the previous example. Single-instruction Element Iterations almost always require more temporary storage than functionally equivalent multiple-instruction Element Iterations.

The previously mentioned "?" function cannot be simulated by any Element Iteration due to the "rule of repeatable pseudo-random number generators": The result of the Roll function is dependent upon the order in which the elements of the result are computed. Thus, the Roll function cannot be simulated by any Iterating Function (or by any Scalar Expression, which goes without saying). Indeed, Roll is not a scalar function at all, since it violates the previously defined Fundamental Criterion of Scalar Functions.

A SCALAR FUNCTION THAT IS TOO COMPLEX

The dyadic "o" scalar primitive (Circular Function) is defined as follows:

```
] XoY          & is  Tanh Y          if X=7
]              &     Cosh Y          if X=6
]              &     Sinh Y          if X=5
]              &     Sqrt 1+YxY      if X=4
]              &     Tan Y           if X=3
]              &     Cos Y           if X=2
]              &     Sin Y           if X=1
]              &     Sqrt 1-YxY      if X=0
]              &     Arc Sin Y       if X=-1
]              &     Arc Cos Y       if X=-2
]              &     Arc Tan Y       if X=-3
]              &     Sqrt (-1)+YxY   if X=-4
]              &     Arc Sinh Y      if X=-5
]              &     Arc Cosh Y      if X=-6
]              &     Arc Tanh Y      if X=-7
]              &     "DOMAIN ERROR" otherwise
```

The APL@CRMS implementation of Circular Function is sort of a hybrid between the Iterating Function and Scalar Expression methods. The Circular Function implementation currently has a front-end which dispatches to:--

--The $8+X^{th}$ SE, if X is a singleton integer between -7 and +7, or

--The Circular Function Iterating Function, if X is non-singleton, or

--The "DOMAIN ERROR" trap generator, otherwise.

Thus, taking the SIN of an entire array,

```
] 1oARRAY
```

causes a dispatch to the SIN, or the $8+1 = 9^{th}$, SE.

At an earlier time, the circular function was implemented as one very long SE, which employed the previously discussed SE conditional branch approximation. This SE was unpardonably inefficient, both time-wise, and space-wise. The time-inefficiency was no surprise, but the space-inefficiency certainly was!!

Dyadic SE / IF Efficiency

Monadic scalar functions are not treated separately in this section because they may be thought of as special cases of dyadic scalar functions: the case where the left operand is a scalar constant. So, many of the conclusions which will be reached about dyadic SE's and IF's apply to monadic SE's and IF's, too. We will be very interested in the relative efficiencies of SE's and their duals.

THE SCALAR OPERAND MICROCOSM

Before tackling the more general case, let's consider the restricted case of scalar-bound source operands. Here, all SE intermediate results are scalars. The same can be said for all IF intermediate results, too. The space required for both implementations is equivalent, although negligible. The APL@CRMS IF implementation, however, necessarily takes more time than its dual, due to the overhead of the IF front-end and back-end parts.

USEFUL TERMS

The phrase, outstanding temps, denotes the maximum amount of space, in words, allocated during a given stage of SE or IF evaluation. Outstanding temps does not include space used for the source operands, as these must be allocated under any algorithm. The phrase, max temps, is MAX(outstanding

temps), taken over all stages in a SE or IF evaluation.

The letters F, S, and R, are the amounts of space used by the first, second, and result operands, respectively. Outstanding temps are often linear functions of F, S and R; such as 2F+3R. Outstanding temps can sometimes be determined syntactically. It is safe to assume that singletons take up negligible space.

IF SPACE-EFFICIENCY

IF's are free to modify their source operands since these are passed by value in APL@CRMS. (There would be no reason to pass these operands by value if APL@CRMS contained instructions to index the I^{th} element of any APL variable, independent of dimensionality. If such instructions existed, the operands could be passed by reference, instead, since they need never be modified by IF's. The SPACE-EFFICIENCY affect of the proposed instructions is mentioned later.) In practice, these modifications always yield variables of approximately the same size as the passed operands. This is because the IF's ravel and reshape passed operands. Once an operand has been ravelled or reshaped, the original operand is discarded. The newly created variable effectively replaces the original operand. Outstanding temps, however, will momentarily grow by the amount of space required by the newly created variable. We will assume for now anyway, that no manipulation is performed <u>in-place</u>.

We will find the outstanding temps for each IF stage, then compute max temps.

Step[1]: Check shapes for conformability. (Outstanding temps = negligible).

Step[2]: Dispatch to one of four Cases:--

Case[A] Generate "RANK ERROR" or "LENGTH ERROR" traps. Exit. No result is returned. (Outstanding temps = negligible).

Case[B] Both operands are singletons. Go to Step[3]. (Outstanding temps = negligible).

Case[C] Both operands are identical non-singleton arrays. Ravel the first operand into a vector. Then ravel the second operand into a vector. Go to Step[3]. (Outstanding temps $\leq$ F=S=MAX(F,S), since each original operand is discarded soon after it is ravelled.)

Case[D] One operand is singleton, the other is not. Ravel the non-singleton. Go to Step[3]. (Outstanding temps $\leq$ R).

Step[3]: Exit if empty result. Initialize the loop counter. Index an element from neither, one, or both vectors depending on whether both, one, or neither source operands are singletons, respectively. Perform an Element Iteration. Store into an element of the result. Update then test the loop counter. Conditionally branch back. (Outstanding temps = negligible, since all values are scalar.)

Step[4]: Reshape the result. Exit. (Outstanding temps $\leq$ R).

Max temps ≤ MAX(R, MAX(F,S)) = MAX(F,S).[20]

But, when Case[C] or Case[D] is entered because one or both non-singleton operand is a vector, a sufficiently observant IF can reduce max temps to negligible, since nothing must be ravelled--as in Case[C] or Case[D]--or reshaped[21] -- as in Step[4]. In fact, the only time max temps has to be as large as MAX(F,S) is when at least one non-singleton source operand has a rank of at least two. This happens very rarely. (Max temps would always be negligable if the proposed index instructions existed.)

SE SPACE-EFFICIENCY

Scalar Instructions used in SE's implicitly use the microcoded dynamic storage allocator to acquire space for their result operands. Scalar instructions also implicitly use the dynamic storage de-allocator to free space held by source operands, but only if they were unassigned[22]

---

[20]Usually, R = MAX(F,S). The exception, R = MIN(F,S), will sometimes occur when one source operand is an empty array, and the other is a singleton. The exception occurs rarely, and when it does, the difference between F and S is small.

[21]The APL@CRMS microprogram could easily be changed so that all no-op reshapes, at least on vectors, are performed in-place. At my urging, the microprogram was changed to ravel a vector in-place.

[22]The APL@CRMS microprogram uses reference counts to keep track how many variables are assigned to the same data.

temporaries. Source operand space is freed only after the scalar instruction's result operand has been allocated and completely computed. Thus, all source and result operands must be simultaneously allocated, briefly. Consider, for example, the real-time space requirements for this SE:

```
]                                F   +   S   +   F   +   F
] & ORDER OF EXECUTION:              3       2       1
] & SPACE FOR SOURCE OPERANDS:     F+R     S+F     F+F
] & SPACE FOR RESULT OPERAND:        R       R       F
] & OUTSTANDING TEMPS:             R+R     R+F     F[23]
Max temps = MAX(R+F,R+R)
```

Parentheses imply stacking. This stacking can increase max temps because the result of the last instruction is no longer always released at the end of the next instruction.

```
]                               (F   +   S)  +   F   +   F
] & ORDER OF EXECUTION:              2       3       1
] & SPACE FOR SOURCE OPERANDS:     F+S     R+F     F+F
] & SPACE FOR RESULT OPERAND:        R       R       F
] & OUTSTANDING TEMPS:             R+F     R+R+F     F
Max temps = R+R+F
```

In this example, inserting parentheses increased max temps.

---

[23]The slash-out means that the source operand's space is to be released at the end of the current instruction.

Sometimes, however, inserting parentheses can decrease max temps:

```
] F+F+S        & max temps = R+R
] (F+F)+S      & max temps = F+R
```

If F were a singleton, and S were a large array, the parenthesized expression above would have lower max temps than the unparenthesized expression.

Definition:--

> To increase a SE's segregation means to delay the intermixing of a SE's operands. As a SE becomes more segregated, more of the intermediate results depend on fewer and fewer source operands.

The second SE of the previous example is more segregated than the first. The next subsection will point out other advantages to segregating the first and second operands.

In conclusion:--

- --Max temps for SE's, unlike IF's, are strong functions of syntax.
- --Most SE's, unlike most IF's, take considerably more than negligible space for max temps.
- --IF's, then, are generally more space-efficient than their duals.

TIME-EFFICIENCY

IF's are generally less time-efficient than their duals. This is mainly because:--

--SE's never ravel or reshape arrays.

--Microcoded loops are by far faster than software loops.

--The number of elements processed in each microcoded loop, unlike an IF loop, is not necessarily as large as the number of elements in the final result: An SE's intermediate results may be smaller than its final result. This very important concept can be exploited by segregating the operands of non-monadic SE's. For example, if A or B, but not both are singletons, and neither are empty,

```
] (A+AxA) + (B+BxB)     & is faster than
] A+B + (AxA) + (BxB)
```

especially when A is singleton, and B is a monstrous array.

The APL@CRMS implementation of the dyadic "⍟" scalar primitive (log to a base), is a SE that uses the monadic "⍟" SE (natural log of):

```
] X⍟Y                   & log, base X, of Y, is implemented as
] (⍟Y) ÷ ⍟X             & which uses the natural logarithm,
]                       & which is, in turn, a very large SE.
```

Not one scalar instruction's result--except for the result of the last division--depends on both X and Y.

The conclusion that SE's are more time-efficient than their duals still leaves an important question unanswered: Are SE's more time-efficient than IF's. The horizontal logic of IF's may contain conditional branches. Thus, it would appear that SE's become less time-efficient with respect to IF's as the amount of conditional logic increases. There is a hard-to-define crossover point, but the APL@CRMS experience has been that only one scalar function contains enough conditional logic to justify being simulated horizontally aq an IF (for solely time-efficiency reasons). That function is the extremely complex Circular Function.

Conclusion

The vertical implementation has been covered much more extensively than the horizontal implementation in this paper. The horizontal implementation, however, is already well-understood and heavily used, whereas the vertical implementation is not. Vertical programming today is rare, except perhaps on array processors and non-backup pipelined processors. Yet, it has been our experience that vertical expressions are both faster and clearer than functionally equivalent horizontal loops.

This paper has been a qualitative study of various aspects of the vertical and horizontal implementations of scalar functions. As such, no single, unifying conclusion has been reached. The various, somewhat-related conclusions which were reached, however, can be relevant to a wide assortment of professions.

Programmers, for example, should try to look upon the vertical, array processing, programming style as a clear and concise representation.

Arithmetic-Unit developers may now be able to justify special case logic to handle frequent multiplications by zero or one.

APL implementers may attach new importance to agreeing to a standard evaluation of pornographic expressions, since multiple assignments can significantly reduce a vertical

program's complexity. Perhaps the APL@CRMS evaluation should be the standard.

Array processor designers, who want to simplify vertical programming, should design a "base set" of scalar instructions which is as complete as possible, yet with a minimal number of hard-to-anticipate properties.

Array processing compiler writers may see the need for an optimizer which maximizes object code segregation by rearranging scalar functions in expressions.

There are many unanswered questions related to this study. A few of these questions which merit further research are:

--How useful are scalar functions with three or more operands? Are monadic and dyadic scalar functions sufficient?

--When should a scalar function be implemented as a blend of the horizontal and vertical methods? The Circular Function is implemented this way.

--Why isn't it obvious when a set of scalar functions is incomplete? Why was it so hard to predict the effect of the new, mixed array, data type?