Steps Toward an APL Compiler

Alan J. Perlis

Research Report #24

January 1974

1. Introduction

APL [1] is the forerunner of an approaching development of array processing languages and machines. Microcircuitry opens the possibility for the design of mini computers that can do direct array processing. Several such designs have already been reported in the literature. APL programming is different from FORTRAN or Algol 60 programming in some important ways. Most importantly the enormous computational potential of the APL "expression" makes the construction of "structured programs" and the efficient execution of these expressions by computer somewhat more difficult than with Algol60 and FORTRAN.

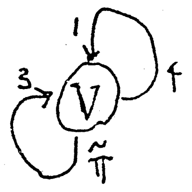
APL is a high-level programming language in the same sense that BASIC, PL/I and the above-mentioned are. But should it ideally translate into the same kind of machine code as these other "scalar" oriented languages? Presumably if one defined an appropriate machine organization APL could be "compiled" into it and the resultant programs run as effectively as machine code programs compiled from FORTRAN. In most implementations APL is handled as a conversational language, and why do otherwise since APL programs tend to be interpreted? However the chief consequence of interpretation is not dynamic creation of text but the dynamic shaping of data arrays--and that dynamic shaping should not be unduly constrained in any compiling effort, e.g., use of declarations to bind shapes. This work has been strongly influenced by the description of an APL machine and various optimization algorithms developed by Phil Abrams in his Ph.D. thesis [2].

2. The ladder

A fundamental activity of APL data is the use of elements of arrays in <u>some</u> order. Hence an array delivers its elements in some sequence. The normal sequence is the ravel, i.e., the subscripts in the order of $(\rho A)^{\circ}.\top\iota\times/\rho A$ in 0 origin for the array A.

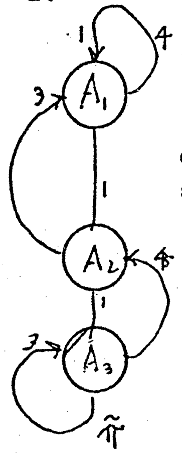
We associate with each use of an array A, a "stream" generator. This generator may be variously represented and implemented, e.g., as an array itself. The generator may be a sequence of generators and such a sequence we will informally denote as a "ladder". Generators are cyclic.

Example: 1. A vector V has the generator:

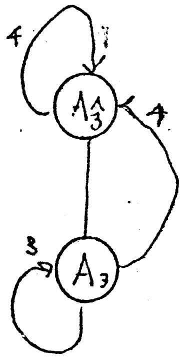


Ⓥ generates the elements of V, denoted by $\hat{\pi}$, in ravel order. $\tilde{\pi}$ is the index or place of $\hat{\pi}$, 1, 3 and 4 represents the initial, "next" and "exit" links of the generator, respectively.

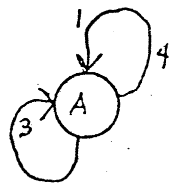
2. An array *A*, ρρ*A* <u>is</u> 3, may have the 3 step ladder.



or the two step ladder



or the one step ladder



$A_{\hat{3}}$ denotes all the generators for *A* but A_3.

We may associate with each use of an array A a "ladder" represented by an array L. To simplify notation we will assume in our examples that ρρ*A* <u>is</u> 3. Then a ladder L for A is:

the array:

PC	β	$\tilde{\pi}$	j	e_{02}
e_{11}	i_1	δ_1	ρ_1	e_{12}
e_{21}	i_2	δ_2	ρ_2	e_{22}
e_{31}	i_3	δ_3	ρ_3	e_{32}

for which ρL _is_ (1+ρρ*A*) , 5. There is _one_ control program that "traverses" any ladder^L and executes the co-routine:

```
    begin
START:        π←β; j←2;
     while    j≤1+ρρA    do begin
              e_j1; L[j;2]←1;
              j←j+1      end;
     while    j≠1        do begin
     while    L[j;2]≤L[j;4] do begin
              e_j2; i_j←i_j+1; π̃←π̃+δ_j      end;
     while    j≤ρρA          do begin
          j←j+1; e_j,1 i_j←1                end end
              j←j-1          end;
              e_0,2;         go to START end
```

the $e_{j,1}$ and $e_{j,2}$ are co-routines and PC points to the place in the ladder control where execution commences (resumes) on use of the ladder control for a particular ladder. β is the base address of the array A and $\tilde{\pi}$ is the current array element location. i_1, i_2, i_3 are the indices in each dimension of the current array value at location.

For ravel sequencing $\delta_1=\delta_2=0$, $\delta_3=1$

Example. let A be 3 2 4 ρ ⍳ 24

A ladder for A is

PC	β	$\tilde{\pi}$	j	e_{02}
e_{11}	i_1	0	3	e_{12}
e_{21}	i_2	0	2	e_{22}
e_{31}	i_3	1	4	e_{32}

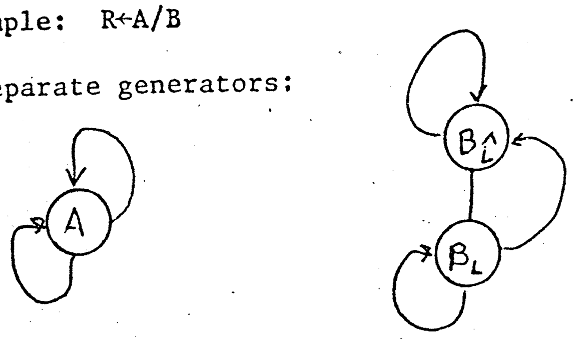
Let e_{32} be print ($\tilde{\pi}$); e_{02} be halt; and the other e_{jk} be null. Then executing the ladder for array A will print out 1, 2, 3, ..., 24.

3. Ladder to ladder communication.

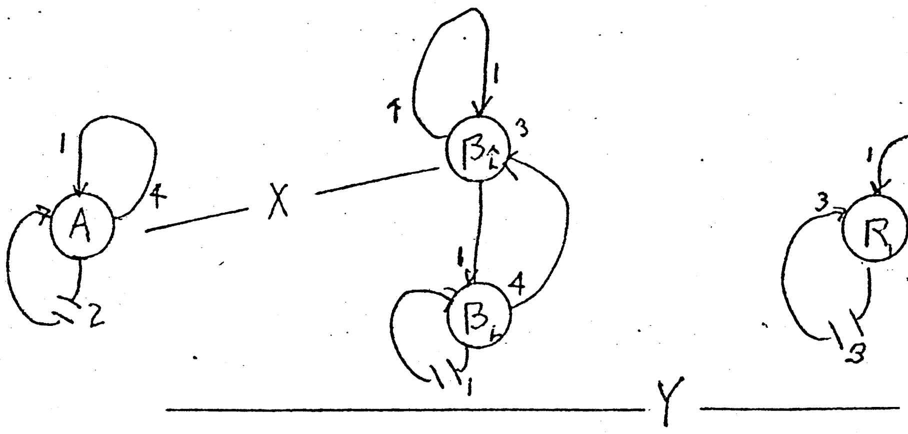
Stream generators communicate by means of co-routine jumps, $\hookrightarrow$ P, each generator being considered a co-routine. Operations on generator outputs are represented as co-routines of sequences of elementary scalar operations.

Example: R←A/B

From the separate generators:



one forms the composite by splicing



1. t←$\tilde{\pi}$; $\hookrightarrow$X

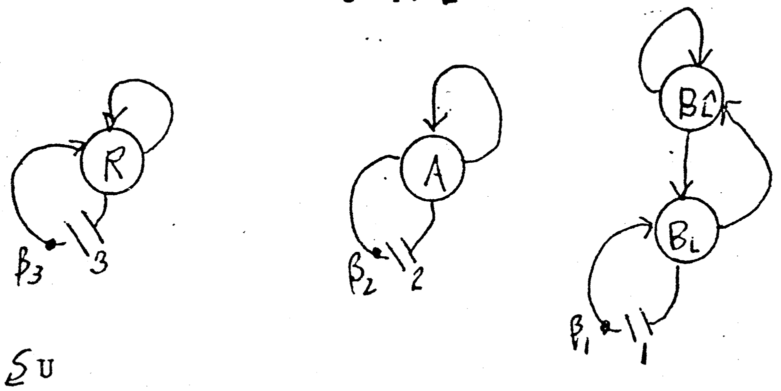
2. <u>if</u> $\tilde{\pi}$ = 1 <u>then</u> $\hookrightarrow$Y ; $\hookrightarrow$X

3. $\tilde{\pi}$←t; $\hookrightarrow$Y

Initially X <u>is</u> 1 <u>of</u> A and Y <u>is</u> 1 <u>of</u> R and control starts at 1 <u>of</u> B

X and Y are co-routine communication registers.

An important transformation of the above stream diagram uses a co-vector. Let U be a co-vector of 2 components, initially set at 1 <u>of</u> A, 1 <u>of</u> R. Then introduce the co-vector shift jump, § U and the preceding diagram may be written as:



1. $t \leftarrow \tilde{\pi}$; § U
2. <u>if</u> $\tilde{\pi} = 1$ <u>then</u> § U <u>else</u> § $1 \phi U$
3. $\tilde{\pi} \leftarrow t$; § U

The vector U attains the following sequence of values:

1 <u>of</u> A, 1 <u>of</u> R

1 of R, β_1

. . .

β_1, β_2

β_2, β_3

....

1 <u>of</u> A, 1 <u>of</u> R .

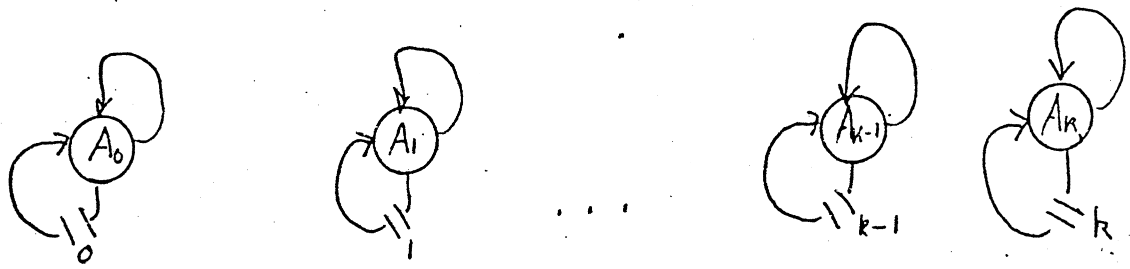
The operation § U has the meaning: Let the control counter contain α. Then

```
$X \leftarrow 1 \uparrow U$
$U[1] \leftarrow \alpha$
$U \leftarrow 1 \phi U$
$\rightarrow X$
```

Suppose we have a sequence of stream generators $S_0, S_1, S_2, \ldots, S_k$ for arrays $A_0, A_1, \ldots, A_k$ with the property that S_j links to S_{j-1}, $j=1,\ldots,k$ and A_0 holds the result of:

$$A_0 \leftarrow A_1 \Theta_1 A_2 \Theta_2 A_3 \Theta_3 \ldots \Theta_{j-1} A_j$$

Where Θ_i are APL dyadic operators. Then: we would have:



with U being initially: 1 <u>of</u> A_{k-1},..,1 <u>of</u> A_0.

4. Operations on ladders

The δ_i control the order of delivery of elements in an array. Some APL operators change this ordering in very regular ways, e.g., ↑,⌽,⍉ and some types of subscripts. In what follows we will often use an array A of rank 3 as a basis for discussion.

$$\tilde{\pi} = M[\pi] = \beta + \sum_{k=1}^{r} (i_k - 1) \times G_k$$

where β <u>is</u> M[1;1;...;1]

The G_k are recursively defined by:

```
G_r ← 1
for k←r-1 step -1 until 1 do
G_k ← G_{k+1} × ρ_{k+1}
```

If we wish to compute the elements, M[π] in ravel order using a ladder of r rungs, passage along any rung increments π by an amount of δ_k on the kth rung. δ_k is the kth component of the vector δ computed by:

```
δ_r ← G_r
for k←r-1 step -1 until 1 do
δ_k ← G_k - ρ_{k+1} × G_{k+1} + δ_{k+1}
```

4.1 Transpose

Suppose we transform A, A'←α⍉A where 1=∧/α∊⍳⍴⍴A AND ⍴α <u>is</u> ⍴⍴A

δ'_3=M'[112]-M'[111]=M[112[⍋ α]-M[111] ⍴A' <u>is</u> α⍉⍴A

Example: *A* <u>is</u> 2 3 4 ⍴ ⍳ 2 4

A'<u>is</u> 312⍉*A* ⍴*A* <u>is</u> 2 3 4

⍴*A*'<u>is</u> 3 4 2

δ'_1=M'[2 1 1]-M'[1 ρ'_2 ρ'_3]=M[2 1 1[3 1 2]]-M'[1ρ'_2 ρ'_3]=M[1 2 1]-M[2 1 4]= -1

δ'_2=M'[1;2;1]-M'[1;1;ρ'_3]=M[1 2 1[3 1 2]]-M[1;1;3[⍋ 3 1 2]]=M[2 1 1]-M[1 3 1]= -11

δ'_3=M'[1;1;2]-M'[1;1;1]=M[2 1 1]-M[1 1 1]= 12

β' = β= 1

and the delivery order is 1 5 9 13 17 21 2 6 ... The formula must be modified if ρ_I=1 for then the corresponding δ_I=0

In general, given δ and G one can compute the δ of the transformed array directly from δ and shape information. But it is easiest to perform the calculations:

Let R← c⍉A where 1= ∧/(⍳⍴⍴A)∊c

then G' <u>is</u> G[⍋ c] and ρ' <u>is</u> ρ[⍋ c]

and

```
$\delta'_r \leftarrow G'_r$
for k←r-1 step -1 until 1 do
$\delta'_k \leftarrow G'_k - \rho'_{k+1} \times G'_{k+1} + \delta'_{k+1}$
```

The diagonal plane case, (⍴c) > ⌈/c, is somewhat more complicated:

To compute ρ':

```
t← ⌈/c; I←1
while t ≥ I do
    ρ'[I]←⌊/(I=c)/⍴A;
    I←I+1
```

To compute G':

```
H← ι0; I←1
while I ≤ t do
  H← H, +/(I=c)/G
  I← I+1
G'← H[⍋((ιρc) = cιc)/c]
```

To compute the new rank: $r' \leftarrow t$

To compute δ' we perform on r', ρ', G' as they have been defined.

Example: R← 3 2 1 2 ⍉ 2 3 4 5 ρ ι 1 2 0

ρ' is 4 3 2

G is 60 20 5 1

H is 60 21 5

G' is 5 21 60

δ' is -97, -39, 60

Note that the δ_k ultimately depend on the shape and the G vector. The latter also makes random accessing quicker.

4.2 Reversal

The operation $\phi[K]A$ reverses the Kth co-ordinate. Thus the Kth co-ordinate is chosen in the order: $\phi\iota\rho[K]$. Let us use the array A' of section 4.1.

Example: $\rho A'$ is 3 4 2, K=1, $\delta_1' = -11$, $\delta_2' = -11$, $\delta_3' = 12$

δ_3''=M'[312] -M'[311]= 12

δ_2''=M'[321] -M'[31ρ_3]= -11

δ_1''=M'[211] -M'[3$\rho_2\rho_3$]= -19

β''=M'[311]= 9

Thus the order of delivery in ϕ[1](3 1 2)⍉A is 9 21 10 22 11 23 12 24 5 17 ...

Indeed, in the case K=1 only δ_1 changes: $\delta_1'' = \delta_1' - 2 \times G_1'$. $\beta'' = \beta + (\rho_1' - 1) \times G_1'$

In case K=2, $\delta_3'' = \delta_3'$, $\delta_2'' = \delta_2' - 2 \times G_2'$, $\delta_1'' = \delta_1' + 2 \times G_2' \times (\rho_2' - 1)$. $\beta'' = \beta + (\rho_2' - 1) \times G_2'$

In case K=3, $\delta_3'' = \delta_3' - 2 \times G_3'$; $\delta_2'' = \delta_2' + 2 \times G_3' \times (\rho_3' - 1)$, $\delta_1'' = \delta_1' + 2 \times G_3' \times (\rho_3' - 1)$.

$\beta'' = \beta + (\rho_3 - 1) G_3''$

The general case for reversal is:

Let $R \leftarrow \phi[K]A$

Let the primes refer to the quantities for R; the unprimed to A:

```
For k ← r step -1 until k+1 do
    $\delta'_k \leftarrow \delta_k$; $G'_k \leftarrow G_k$
$\delta'_k \leftarrow \delta_k - 2 \times G_k$; $G'_k \leftarrow -G_k$
for k ← k-1 step -1 until 1 do
$\delta'_k \leftarrow \delta_k + 2 \times G_k \times (\rho_k - 1)$;
$\beta' \leftarrow \beta + G_k \times (\rho_k - 1)$
```

4.3 Drop (↓)

Consider $(Q_1\ Q_2\ Q_3) \downarrow A$

where A has the shape $\rho_1\ \rho_2\ \rho_3$

$M'[I;\ J;\ K\ ] = M[I + (Q_1 > 0) \times Q_1;\ J + (Q_2 > 0) \times Q_2;\ K + (Q_3 > 0) \times Q_3]$

and ρM; is $\rho M - |\ (\ Q_1\ Q_2\ Q_3)$ For example,

$\delta'_1 = M'[2;1;1] - M'[1;\rho_2;\rho_3]$ while $\delta'_3 = \delta_3$

From which we can obtain: $\delta'_1 = \delta_1 + (|Q_2) \times G_2 + (|Q_3) \times G_3$

$\delta'_2 = \delta_2 + (|Q_3) \times G_3$, $\delta'_3 = \delta_3$

$\beta' = M'\ [1;1;1]$

The general situation for $Q \downarrow A$ is:

```
$\delta'_r \leftarrow \delta_r$;t ← 0
for k ← r-1 step -1 until 1 do
    t ← t + $(|Q_k) \times G_k$
    $\delta'_k \leftarrow \delta_k + t$
$\beta' \leftarrow \beta + \sum_{k=1}^{r} (Q_k > 0) \times Q_k \times, G_k$
```

4.4 Take (↑)

$(Q_1\ \ Q_2\ \ Q_3) \uparrow A$ and ρA <u>is</u> $\rho_1\ \ \rho_2\ \ \rho_3$.

Then ρ' <u>is</u> $|\,Q_1\ \ Q_2\ \ Q_3$

$M'[I;J;K]=M[I+ (Q_1 < 0) \times |\, \rho_1-Q_1;\ J + (Q_2 < 0) \times |\, \rho_2-Q_2;\ K + (Q_3 < 0) \times |\, \rho_3-Q_3]$

Using the definitions of δ_i we can derive:

$$\delta_1' = \delta_1 + (\rho_2 - |Q_2) \times G_2 + (\rho_3 - |Q_3) \times G_3$$

$$\delta_2' = \delta_2 + (\rho_3 - |Q_3) \times G_3\ ,\ \delta_3' = \delta_3$$

$$\beta' = M'[1;1;1] = \beta + ((Q_1 < 0) \times |\, \rho_1 - Q_1) \times G_1 + ((Q_2 < 0) \times |\, \rho_2 - Q_2) \times G_2$$
$$+ ((Q_3 < 0) \times |\, \rho_3 - Q_3) \times G_3$$

The general situation for $Q \uparrow A$ is:

```
$\delta_r' \leftarrow \delta_r; t \leftarrow 0$
for $k \leftarrow r-1$ step $-1$ until $1$ do
    $t \leftarrow t + (\rho_k - |Q_k) \times G_k$
    $\delta_k' \leftarrow \delta_k + t$
$\beta' \leftarrow \beta + \sum_{k=1}^{r} ((Q_k < 0) \times |\, \rho_k - Q_k) \times G_k$
```

The G vector is not affected by ↑ and ↓.

4.5 Rotate

The general rotation operator is:

$$N\ \phi\ [K]\ A$$

where ρA <u>is</u> $\rho_1\ \rho_2 \ldots \rho_K \ldots \rho_{\rho\rho A}$, ρN <u>is</u> $(K \neq \iota\rho\rho A)\ /\ \rho A$

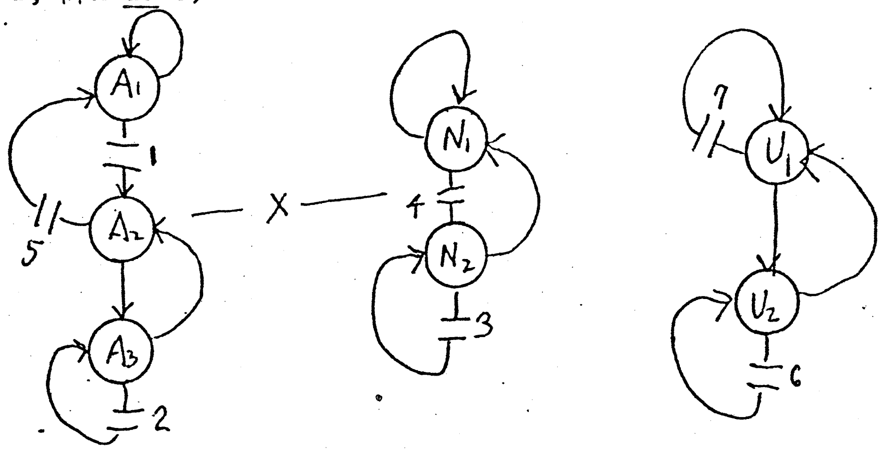
Example: A <u>is</u> 3 2 4 ρ ι 24 , K <u>is</u> 2

N <u>is</u> 3 4 ρ 1 0 1 2 1

Define $\nabla_3 = 1,\ \nabla_I = \rho_{I+1} \times \nabla_{I+1}\ \ I = 1,2.$

The delivery order is 5, 2, 7, 4, 1, 6, 3, 8, 13, 14 ...

The general rotation operator does not compose like ⍉, ↓, ↑ and ⌽ [*K*] and it is an example of a "break" operator, i.e., one that induces temporary storage. If the operators to the left of ⌽ [*K*] in an expression do not alter delivery orders the temporary storage array U required is of shape $\rho_k\ \rho_{k+1} \cdots \rho_{\rho\rho A}$ for we may deliver that many elements from *A* to U, rotate within U, and deliver the rotated elements as the result. The stream network, in such a case is: (shown for the case k=2, ρρ*A* <u>is</u> 3):



X: initially 1 <u>of</u> N_1; Z: initially 1 <u>of</u> destination; Y: initially 1 <u>of</u> U

1. $u \leftarrow I_1$
2. $t \leftarrow \nabla_1 | \tilde{\pi} - 1$; $q \leftarrow M[\tilde{\pi}]$; ↪ X
3. $S \leftarrow M[\hat{\pi}]$; $h \leftarrow \nabla_1 | t - S \times \nabla_2$; $U[h] \leftarrow q$; ↪ X
4. $\tilde{\pi} \leftarrow \beta + (u-1) \times \nabla_1$
5. ↪ Y
6. $w \leftarrow M[\tilde{\tilde{\pi}}]$; ↪ Z
7. ↪ Y

4.6 Subscripts

APL permits very flexible subscripting. We will limit our attention to arrays each of whose subscripts is (a) blank, (b) scalar constant or (c) an interval (of the form c:d as in Algol 60). Subscripting is reducible to a case of take and drop applied successively.

Thus $X[M_1;M_2;M_3]$ is $(Q_1\ Q_2\ Q_3) \uparrow (P_1\ P_2\ P_3) \downarrow X$

where: if M_i is blank, $Q_i = \rho_i$, $P_i = 0$

if M_i is the constant c then

$Q_i = 1$ and $P_i = c - 1$

if M_i is the interval c:d then $P_i = c - 1$ and Q_i <u>is</u> d-c+1.

Thus we have shown that a sequence of the operators ⍉, $\phi[K]$ (reversal), ↑, ↓ and [;;] of a restricted type can be collapsed into one calculation of the vectors δ, ρ, G and the constant β. Clearly it is appropriate that such sequences be collapsed prior to the evaluation of expressions. Let us now extend these simplifications to include other operators.

4.7 Reduction

Consider $\Theta/[2]A$ where Θ is a scalar dyadic operator and where ρA <u>is</u> $\rho_1\ \rho_2\ \rho_3$ then this is equivalent to: $\Theta/(1\ 3\ 2)$⍉A. In the general case $\Theta/[k]A$, where ρA is $\rho_1\ \rho_2 \ldots \rho_N$ is equivalent to $\Theta/(1\ 2\ \ldots N\ k{+}1\ N{-}1)$⍉1.

Furthermore,

c⍉Θ/A is equivalent to $\Theta/(c;\rho\rho A)$⍉4

c ↑ $\Theta/A \equiv \Theta/(c;(\rho A)[\rho\rho A])$ ↑ A

c ↓ $\Theta/A \equiv \Theta/(c;0)$ ↓ A

$\phi[K]\Theta/A \equiv \Theta/\phi[K]\ A$

and $(\Theta/A)[M_1;M_2;\ldots,M_N] \equiv \Theta/A[M_1;M_2;\ldots M_N]$.

4.8 Outer product $\quad A \overset{\circ}{.} \Theta B$

Consider the case:

c⍉$A \overset{\circ}{.}\Theta B$

We introduce the notation

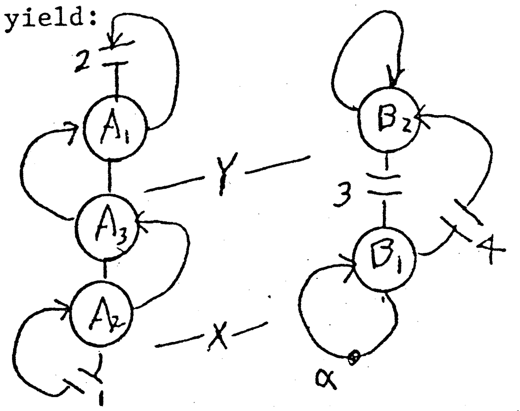
{c⍉°.} to specify an order in which the elements of A and B are to be combined.

Consider the following examples:

(a) ρA is 3 4 5 ; ρB is 4 6

A{2 4 3 5 1 °.}+B

Then the streams for A and B are spliced after transpositions: 1 3 2⍉A and 2 1⍉B to yield:



joined with co-routine register initialized to 1 of A , X initialized to 1 of B_1.

The code pieces are:

1. t←M[π]; ↪X
2. ↪Y
3. ↪Y
4. ↪X

A splice at α would give:

r←tΘ M[π]; ↪ w where w is the co-routine register connecting to the user of the result.

(b) A and B as in (a) and

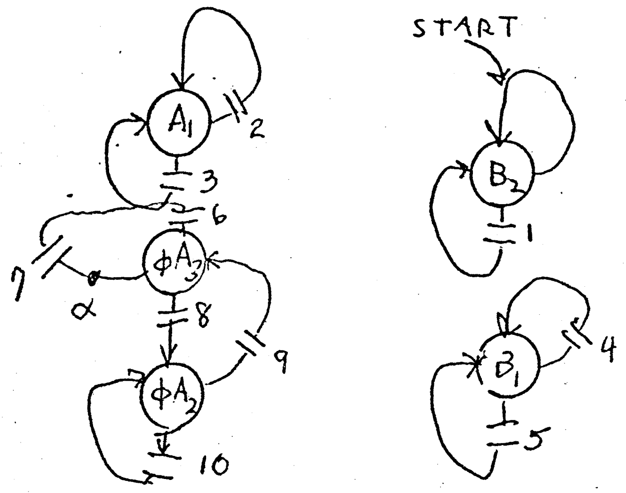
R←(2 3 1)⍉ ×/[2] + /[2]((3 2 1)⍉A)°.+B

This is equivalent to

```
  R ← (2 3 1)⍉×/(1 4 2 3)⍉...
≡ R ← ×/(2 3 1 4)⍉(1 4 3 2)⍉+/[2]...
≡ R ← ×/(2 4 1 3)⍉+/[2]...
≡ R ← ×/(2 4 1 3)⍉+/(1 5 2 3 4)⍉ ..
≡ R ← ×/+/(2 4 1 3 5)⍉(1 5 2 3 4)⍉(...
≡ R ← ×/+/(2 5 4 1 3)⍉(3 2 1⍉A)° . + B
≡ R ← ×/+/(3 2 1⍉A){2 5 4 1 3° .}+ B
```

yielding:

```
≡ R ← ×/+/φ̃[3]φ̃[2](3 2 1)⍉A{ 2 5 4 1 3° .}+B
```



1. ↪ Y
2. ↪ Y
3. ↪ Z
4. ↪ Z
5. r ← M[$\tilde{\pi}$]; ↪ X
6. S ← 1
7. ↪ X
8. t ← 0
9. S ← t×S
10. t ← t + M[$\tilde{\pi}$]+r

α. to destination ↪ W

Initially

X is set to 1 <u>of</u> A_3

Y is set to 1 <u>of</u> A_1

Z is set to 1 <u>of</u> B_1

The network commences at 1 <u>of</u> B_2

Consider the case:

$(q_1\ q_2\ q_3\ q_4\ q_5) \uparrow A^{\circ}.\Theta B$

Again we write as

$A\{(q_1\ q_2\ q_3\ q_4\ q_5) \uparrow {}^{\circ}.\}\Theta B$

which becomes:

$((q_1\ q_2\ q_3)qA)^{\circ}.\theta(q_4\ q_5) \uparrow B$

↓, [;;] and φ[K] similarly decompose.

4.9 Inner product.

The inner product is a specialized use of outer product, reduction and transposition. The following definition holds:

if $R \leftarrow A\ \Theta_1 .\ \Theta_2\ B$ is defined then so is

$\Theta_1/\quad$ Ƶ⍉$A^{\circ}.\ \Theta_2 B.$

where Ƶ is $(\iota\text{-}1 + \rho\rho A)$, $(2\rho\text{-}1 + (\rho\rho A) + \rho\rho B)$, $(\text{-}1 + \rho\rho A) + \iota\text{-}1 + \rho\rho B$

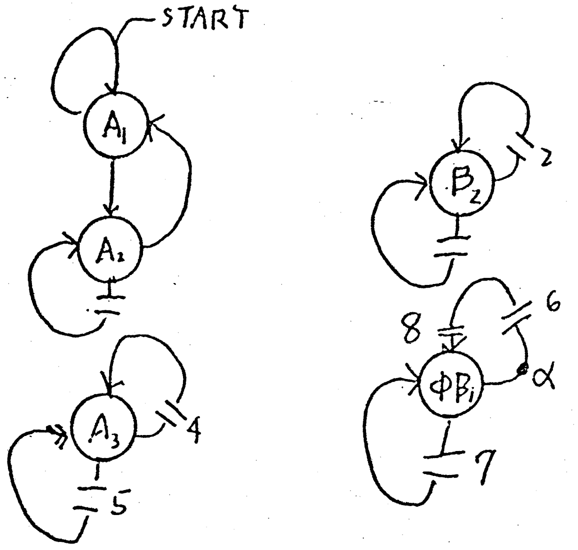
and is equal to R. Indeed this is exactly how the inner product operator is implemented from the streams for A and B, i.e., as:

$\Theta_1/$ A{Ƶ⍉$^{\circ}$.}Θ_2B

Thus, are all the simplifications associated with the outer product adapted to the inner product.

Example: $\rho\rho A$ <u>is</u> 3, $\rho\rho B$ <u>is</u> 2.

Θ_1/A {1 2 4 3⍉$^{\circ}$}.Θ_2B



1. $\hookrightarrow \overline{X}$
2. $\hookrightarrow \overline{X}$
3. $\hookrightarrow Y$
4. $\hookrightarrow Y$
5. $t \leftarrow M[\pi]; \hookrightarrow Z$
6. $\hookrightarrow Z$
7. $S \leftarrow (t\Theta_2 M[\tilde{\pi}])\Theta_1 S$
8. $S \leftarrow$ identify (Θ_1)

α. result is S; $\hookrightarrow W$

Initially: $\overline{X}$ <u>is</u> 1 <u>of</u> B_2
Y <u>is</u> 1 <u>of</u> A_3
Z <u>is</u> 1 <u>of</u> B_1

For each of the selection operators Ω we have the identities:

4.10 Index of. (dyadic ⍳):

Ω A ⍳ B ≡ A ⍳ Ω B

4.11 Membership. (∊):

A ∊ B ≡ (Ω A) ∊ B

4.12 Compression /:

Ω A/[K]B

(a) Ω <u>is</u> c⍉

c⍉A/[K]B ≡ A/c[1 2...ρρB...-1 + ρρB] ⍉ B

(b) Ω is c↑ <u>or</u> c↓

c↑ A/B ≡ A/(c,-1↑ρB) ↑ B

c↓ A/B ≡ A/(c,0) ↑ B

(c) ⌽[K] A/B ≡ A/⌽[K]B , K≠ρρB

≡ (⌽A)/⌽B , K=ρρB or elided.

4.13 Expansion \ :

Treated in precisely the same way as compression.

4.14 ⍋(and ⍒)

(a) A⍉⍋[K]B ≡ ⍋[A[K]]A⍉B

However

⌽[K]⍋[J]B ≡ ⍋[J]⌽[K]B only if K≠J [;;], c↑ and c↑ do not usefully commute with ⍋ and ⍒ .

4.15 Catenate

cϕA,[K]B ≡ (cϕA), [c[K]] cϕB

where (ρρB) = ρρA and 1 = ∧/(K ≠ ιρρA)/(ρA) = ρB

If (ρρB) ≠ ρρA and |(ρρA) − ρρB| = 1 and 1 = ∧/(ρB) = (K≠ιρρA)/ρA then

cϕA,[K]B ≡ (cϕA), [c[K]]((K ≠ ιρρA)/c)ϕB

If (ρρB) ≠ ρρA and B is a scalar, α then create the array B' <u>is</u>

((K≠ιρρA)/ρA)ρ α, reduce to the case cϕA,[K]B'.

For ϕ[K]:

ϕ[K] A,[J]B ≡ (ϕ[K]A), [J] ϕ[K]B K ≠ J

≡ (ϕ[K]B), [K] ϕ[K]A K = J

For c ↑ A,[K]B:

≡ (c'↑A), [K] D'↑B

In case (i):

c' <u>is</u> obtained from c by changing c[K].

c[K] ← ((c[K]≥0)× ((ρA)[K])⌊c[K]) + (c[K]< 0)×((c[K]) + ρB)⌊0

and D' from c by changing c[K]:

c[K] ← ((c[K]≥0) × (c[K]−(ρA)[K])⌈0) + (c[K]< 0) × c[K]⌈−(ρB)[K]

In case (ii):

c[K] ← ((c[K] ≥ 0) × ((ρA)[K]⌊c[K]) + (c[K]< 0) × ((c[K])+ 1) ⌊0

But D' is now:

D' ← (c[K]≥ 0)∧((c[K]) ≤ (ρA)[K]) × (ρB)ρ0

+((c[K]>(ρA)[K]) ∨ c[K]< 0) × (K ≠ ιρA)/c

We omit the computations for c ↓ A,[K]B since they are so similarly derived.

4.16 Example.

Let ρA be 2 3, ρB <u>be</u> 2 3, ρc <u>be</u> 3 2 and the expression to be evaluated:

```
2 ¯1 1 ↑ A + .× 3 1 2⍉+/[1]B °. +c
≡           ...+/B{4 1 2 3⍉°.}+c
        +/3 1 2 4⍉B{4 1 2 3⍉°.}+c
           +.×+/B{4 3 1 2⍉°.}+c
           +̃.× 3 1 2⍉ +/B{4 3 1 2⍉°.}+c
           +̃.×+/ 3 1 2 4⍉B{4 3 1 2⍉°.}+c
     (2 ¯1 1) ↑ A +̃.×+/B{4 2 3 1°.}+c
       (2 3 ↑ A)+̃.× ¯1 1 3 ↑ +/B{4 2 3 10.}+c
           ...+/ ¯1 1 3 2 ↑ B {4 2 3 1°.}+c


       (2 3 ↑ A) +̃.×  +/ (2 1 ↑ B){4 2 3 1°.}+(3 ¯1) ↑ c
```

where X +̃.× Y is an inner product where last of X is combined with last of Y.

5. Rags

A useful generalization of the APL array is the ragged array. A ragged array is "uneven" only in its last subscript position. For ragged arrays the operator α plays a role analogous to that of ρ. Thus, let H be an array and V a vector. HαV shapes V into a ragged array such that the value of an element of H is the length of the corresponding rag of V. For example:

```
H is 7 9 11
W ← Hα 'THE ONE AND ONLY RAG PICKER'
W[2;] is 'AND ONLY'
The unary α gives the rag shape of W.
        αW is 7 9 11
```

The unary α gives the rag shape of W:

αW <u>is</u> 7 9 11

An array A can, of course, be made ragged:

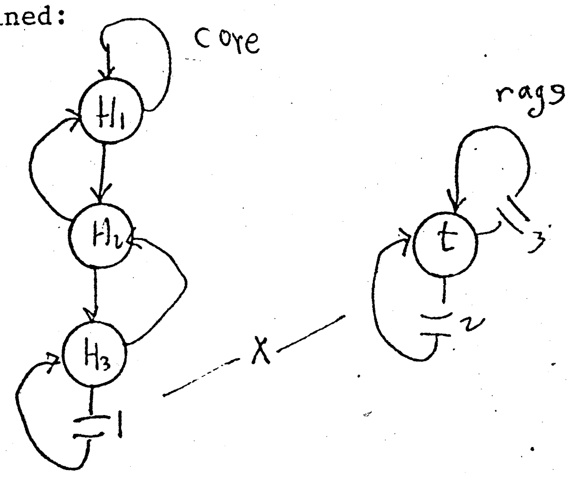
A ← ((¯1 ↓ ρA) ρ ¯1 ↑ ρA)αA

αW is the null vector if W is not ragged. Ragged arrays are homogeneous.

Ragged arrays provide a convenient means of attaching a single name to a collection of vectors. For example, an APL function can be looked upon as a ragged array whose associated "rags" are the header and function lines. There is some advantage to being able to represent functions as a more structured data type than the character string. While ragged arrays do not provide the most general data structure one might like, they do provide additional flexibility to APL without doing violence to the semantics and syntax of the language.

Suppose M is a ragged array for which ρραM <u>is</u> k. Then k+1 subscripts are needed to isolate an entry of M. Suppose k <u>is</u> 2. Then M[I;J;] is the rag having (αM) [I;J] elements. M[;;1] is the array of first components of all the rags of M. The generalization of V[Q], where V and Q are vectors and 1=∧/Q∊ιρV, to ragged arrays is W[H] where W and H are ragged arrays for which αW and αH are identical and for I_1, I_2,...I_k such that I_p ∊ ι (ραW)[p], p=1, 2,...,k
1 = ∧/ H[I_1;...;I_k;] ∊ ιαW[I_1;I_2;...;I_k].

The control for the access of ragged arrays is through the use of two ladders, called the "core" and the "rag". The core produces, in order, the location from which the pair β_k, ρ_k used in successive deliveries by the rag, can be determined:



1. $t \leftarrow M[\tilde{\pi}];\hookrightarrow X$
2. $S \leftarrow M[\tilde{\pi}]$
3. $\hookrightarrow X$

where t is the rag base and knowing t we may computer ρ of the rag base.

Operations on αW and the rags are treated as described in sections 4.2 to 4.6.

Ragged arrays are most conveniently treated as delivering a succession of arrays of rank 1 (vectors) and thus, with their inclusion, some APL operators can be extended to become vector operators.

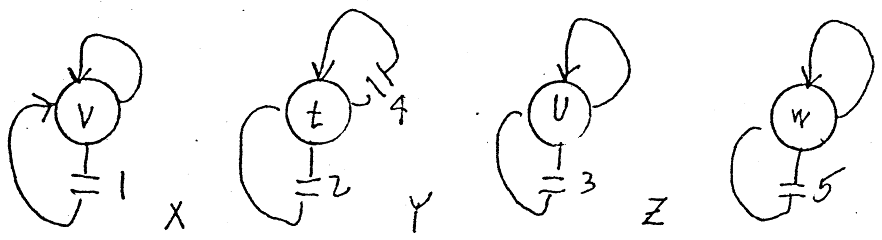
These extensions are simply handled through the use of ladder networks, but not so easily handled in ordinary APL syntax. We give two examples:

(i) H ← (1 +(M ≠ ' ')ι1)⌽M

left justifies the character string vector M. Suppose however M is a ragged character array. One would postulate that the same action is to hold on successive rags of M.

(ii) V[⍋V]

sorts the vector V. But then what interpretation are we to give (ii) when V is a ragged array? The purpose of ragged arrays is to attach a sequence of vectors to the same name so that similar actions on all members of the sequence can be easily and naturally programmed. Consider example 2, expressed in stream form



1. $t \leftarrow M[\pi];\hookrightarrow X$
2. $S \leftarrow M[\pi];\hookrightarrow Y$
3. $M[\pi]\leftarrow S;\hookrightarrow Y$
4. $W \leftarrow$ ⍋ $U;\hookrightarrow Z$
5. $M[\pi] \leftarrow$ base of $W;\hookrightarrow X$

To maintain simple syntax it is to be understood that ragged arrays occurring in an expression are all synchronously being delivered (in their ravel order) one vector (instead of one scalar) at a time. Thus

```
M[⍒M], T[⍒T ← 2 ↓ M]
```

deliver ragged arrays (in sequenced orders) whose rags are individually sorted. This interpretation defines the extension of operators to ragged arrays. More specifically, if V, V_1 and V_2 are ragged arrays:

1. Monadic and dyadic scalar operations Θ have the extensions; $V_1 \Theta V_2$ is defined if either V_1 or V_2 is a scalar or an array of one element or if $\alpha V_1 = \alpha V_2$ or if $\rho\alpha V_1 = \rho\alpha V_2$ and if $(\alpha V_1)[I_1;\ldots I_k] \neq (\alpha V_2)[I_1;\ldots;I_k]$ then at least one of $(\alpha V_1)[I_1;I_2\ldots,I_k]$ and $(\alpha V_2)[I_1;\ldots I_k]$ is 1. Then the operation Θ is applied to corresponding rags of V_1 and V_2 as a succession of vector operations.
2. The core of a ragged array may be thought of as holding, in each array position, <u>two</u> data: ρ of the corresponding rag <u>and</u> the corresponding rag. Thus Θ/V applies to the rags of V and produces an array.
3. Inner and outer products are not defined when one of the operants is ragged.
4. The index generator, monadic ι is extended to vectors: ιV is a ragged array, e.g., ιι4 is

```
1
1 2
1 2 3
1 2 3 4
```

5. Ravel is the ravel of the ravel of the rags.
6. Reversal ⌽V is the reversal of the rags of V.

   ⌽[K]V, K≠1+ρρα V refers to reversals of the core.
7. Monadic transposition does not apply to a ragged array.
8. Grade-up and-down apply to the rags.

9. Reshape, in the form HαV, restructures ragged arrays. The following is a "crop" by a ragged array V to a homogeneous array:

((αV), n)ρn ↑ V

10. Catenation, V_1, V_2 is defined if $(\alpha V_1) = \alpha V_2$ and the corresponding rags are catenated. However a more general definition of catenation for ragged arrays is possible and is based on treatment of array catenation in some APL implementations. V_1,[K]V_2 is permitted and has the significance of concatenating αV_1 , αV_2 in the K direction. The conformability conditions are:

(1) If $(\rho\rho\alpha V_1) = \rho\rho\alpha V_2$ then K must be in $\iota\rho\rho\alpha V_1$ and $(\rho\alpha V_1) = \rho\alpha V_2$ except possibly in the K dimension.

(2) If $(\rho\rho\alpha V_1) \neq \rho\rho\alpha V_2$ then $1 = (\rho\rho\alpha V_1) - \rho\rho\alpha V_2$ or αV_2 is an array of rank 1., and $(\rho\alpha V_1) = \rho\alpha V_2$ except for the K co-ordinate.

Examples:

(a) αV_1 <u>is</u> 2 3 ρι6

αV_2 <u>is</u> 4 3 ρ3+ι9

$\rho\ \alpha V_1$,[1]V_2 <u>is</u> 6 3 and

$\rho(V_1$,[1]$V_2)$[5;2;] <u>is</u> 11

(b) $\rho\alpha V_1$ <u>is</u> 5 3

$\rho\alpha V_2$ <u>is</u> 5

then $\rho\ \alpha V_1$,[2]V_2 <u>is</u> 5 4

11. Rotation. Rotation may be on the rags part of a ragged array, V. The rags may be rotated separately as in AφV where (ρA) = ραV. However the core may also be rotated as in use of Aφ[K]V where K ≠ 1 + ρραV.

12. Index of (dyadic ι). Let V_2 be a ragged array. If, in $V_1 \iota V_2$, V_1 is a vector then $V_1 \iota V_2$ has the same interpretation as if V_2 were a standard array. If, however, V_1 is itself a ragged array then $(\rho\alpha V_1) = \rho\alpha V_2$ and the operation is applied to corresponding rags.

Example:

```
        1 2          3 4 2 1             3 3 2 1

        1 2 3   ι    1 1 3 2     is      1 1 3 2

        2 1          2                   1

        4            4 4 5               1 1 2
```

13. Base value (decode). In $V_1 \bot V_2$, if V_1 is a vector then the original APL decode is applied (with V_1) to every rag in V_2. If V_1 is a ragged array then $(\rho\alpha V_1) = \rho\alpha V_2$ and decode is applied to corresponding rags.

14. Representation (encode). In $V_1 \top V_2$ is V_2 is a vector, then V_1 and encode is applied to each component of V_2. If V_1 is a ragged array then V_2 must be an array such that $(\rho\alpha V_1) = \rho V_2$, or V_2 must be a scalar.

15. Compression. Compression may be applied to each of the rags as in V_1/V_2 or to the core of V_2 as in $V_1/[K]V_2$ for $1 \le K \le 1 + \rho\rho\alpha V_2$.

16. Expansion. Rags may be expanded if they obey the conformability condition: In $V_1 \backslash V_2$, $(+/V_1[I;J;]) = (\alpha V_2)[I;J]$

17. Dyadic transposition is applicable only to the core of a ragged array.

18. Take ↑ and drop ↓ are applicable to both the core and the rags of a ragged array.

19. Membership. If V_1 and V_2 are ragged arrays the membership function is applied rag by rag.

With respect to scalar monadic and dyadic operations scalars are permitted to combine with arrays, as in

$3 + 2\ 2\rho\iota 4$ being $2\ 2\rho 3 + \iota 4$.

Ragged arrays provide a data structure which permits vectors (arrays of rank 1) to combine with ragged arrays using many of the standard APL operators. Thus in U⊖V if V is a ragged array then U is either a vector or a ragged array for which ρι U and ρι V are the same. In the former case U as a left operant is combined with each rag of V, while in the latter corresponding rags of U and V are combined. This conformability condition applies to the operations:

ϵ ,(dyadic)ι, ⊥, (dyadic)/, (dyadic)\

In subscripting, if U and V are ragged arrays and obey the above conformability condition V[U] applies each rag of U to the corresponding rag of V to select.

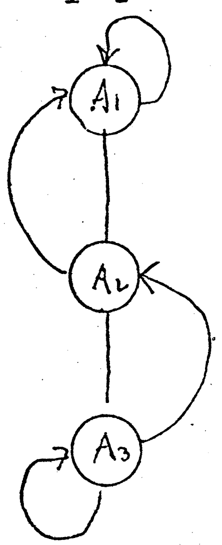
20. Indexing is applied to core and/or rags.

6. Operator definition in terms of ladder networks.

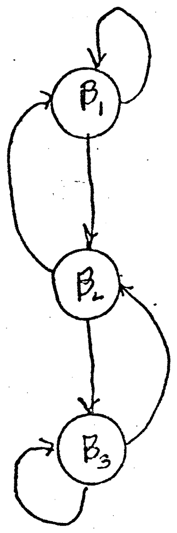
Given a set of arrays each with its own stream generator, how are they combined under APL operations to yield the stream generator for an expression? Five pieces of information are required from generators in order to "splice" them together:

1. entry point
2. exit point
3. result
4. emission point
5. a control communication or synchronization register.

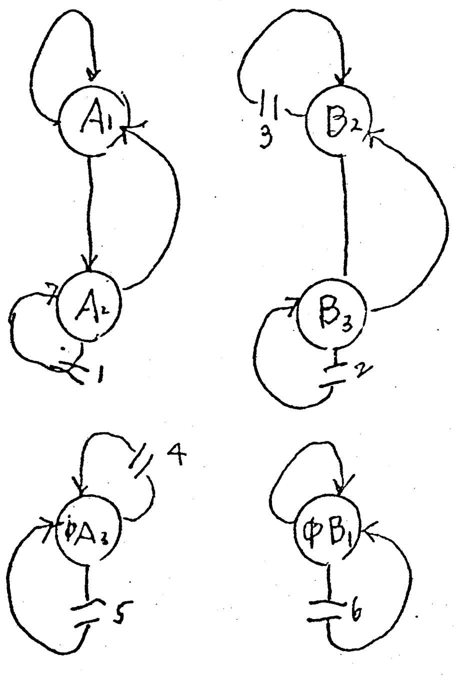
Let us consider an example: Given A and B, construct the generator for A $f_1.f_2$ B. Assume ρρA <u>is</u> 3, ρρB <u>is</u> 3. From



and



we construct



with the attached code:

1. $\hookrightarrow$ X
2. S ← identify (f_1); $\hookrightarrow$ Y
3. $\hookrightarrow$ X
4. <u>result</u> S; $\hookrightarrow$ q; $\hookrightarrow$ Y
5. t ← M[$\tilde{\pi}$]; $\hookrightarrow$ Z
6. S ← (t f_2 M[$\tilde{\pi}$]) f_1 S; $\hookrightarrow$ Z

where the new stream has:

1. entry point: entry point of A
2. exit point: exit point of A
3. result : S
4. emission point: code piece 4
5. synchronization register: q

Let us carry the process one step farther. Suppose the expression is:

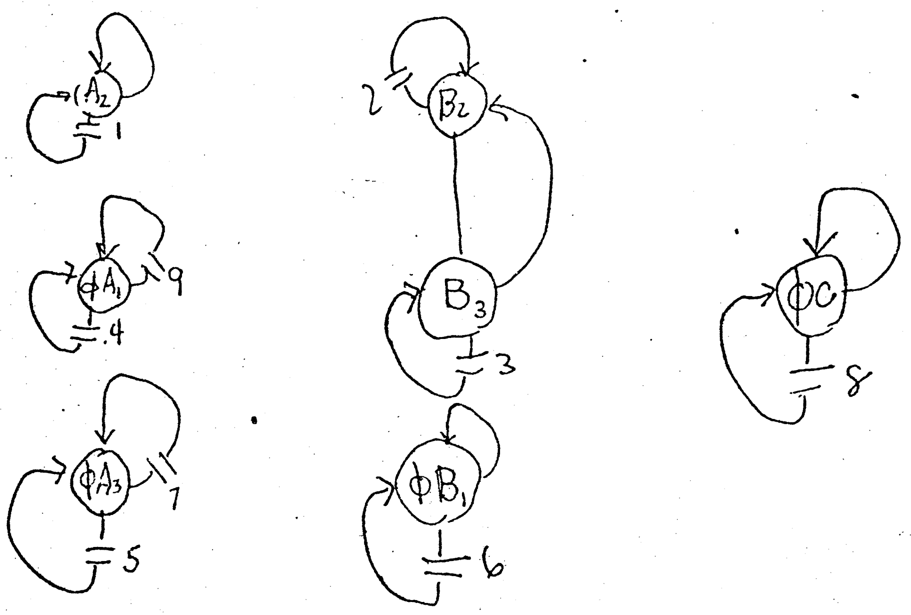
$Cf_3.f_4Af_1.f_2B$ where ρρC <u>is</u> 1 and A and B are as before. Then the expression reduces to:

$$Cf_3\overset{\sim}{.}f_4(4\ 1\ 2\ 3)⍉Af_1\{1\ 2\ 5\ 6\ 3\ 4.\}f_2B$$

and then to

$$Cf_3\overset{\sim}{.}f_4Af_1\{2\ 5\ 6\ 1\ 3\ 4\ \overset{\sim}{.}\}f_2B$$

and the stream network with attached code is:



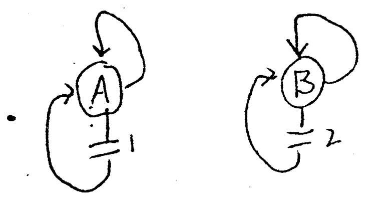
```
1. $\hookrightarrow$ X
2. $\hookrightarrow$ X
3.  t ← indentify ($f_3$); $\hookrightarrow$ Y
4.  r ← identify ($f_1$); $\hookrightarrow$ Z; $\hookrightarrow$ V
5.  S ← M[$\tilde{\pi}$]; $\hookrightarrow$ W
6.  r ← (S $f_2$ M[$\tilde{\pi}$]) $f_1$ r; $\hookrightarrow$ W
7.  $\hookrightarrow$ Z
8.  t ← (M[$\tilde{\pi}$] $f_4$ r) $f_3$ t; $\hookrightarrow$ V
9.  result t; $\hookrightarrow$ U; $\hookrightarrow$ Y.
```

The quintet for this stream is: entry of A_2, exit of A_2, t, q, V

The initial values of the internal communication registers X, Y, Z, V and W are the entry points to θ_2, A_1, A_3, C and B_1.

We now proceed to an examination of the individual APL operations.

1. Scalar operations A θ B

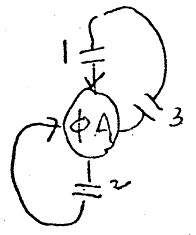


1. $t \leftarrow M[\tilde{\pi}];\hookrightarrow X$

2. $r \leftarrow t\ \theta\ M[\tilde{\pi}];\hookrightarrow q;\hookrightarrow X$

{ entry of A, exit of A, r, 2, q} init (X) <u>is</u> 1 <u>of</u> B

2. Reduction: θ/A



1. S ← identify (θ)

2. $S \leftarrow M[\tilde{\pi}]\ \theta S$

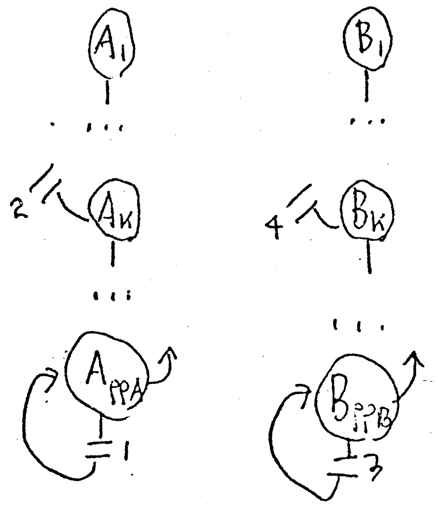
3. $\hookrightarrow q$

{ entry of A, exit of A, S, 3, q}

3. The inner product has already been described.

4. Catenation: A,[K]B

(1) (ρρA) = ρρB



1. $S \leftarrow M[\tilde{\pi}];\hookrightarrow q$

2. $\hookrightarrow X$

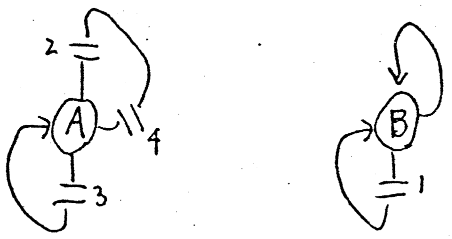
3. $S \leftarrow M[\tilde{\pi}];\hookrightarrow q$

4. $\hookrightarrow X$

{ entry of A, exit of B, S,(1,3);q} init (X) <u>is</u> 1 <u>of</u> B

(ii) (ρρA) ≠ ρρB. The same stream structure as (i) holds

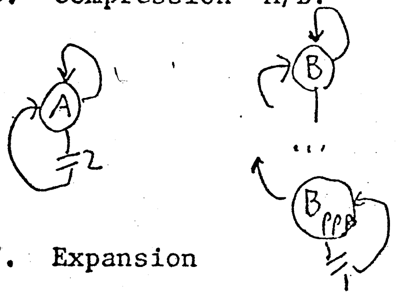
5. Index (A ⍳ B)



```
1. S ← M[π̃];↪X
2. k ← 1; r ← 1 + ⍴A
3. if M[π̃]=S then r ← k; go to 4
   else k ← k+1
4. ↪q;↪X
```

{entry of B, exit of B, r, 4, q} init (X) <u>is</u> 1 of A.

6. Compression A/B.



```
1. S ← M[π];↪X
2. if M[π] = 1 then ↪q; ↪ X
```

{entry of B, exit of B, s, 2, q} init (X) <u>is</u> 1 <u>of</u> A.

7. Expansion

The same as for compression except in the code pieces:

```
1. S ← M[π];↪X
2. if M[π] = then S ← unit(B);↪q;↪X
```

7. Expression evaluation

We will now consider an example where all the ranks are known prior to execution of the expression:

```
R ← ×/[1](3 4 1 2)⍉+/[2](1 3 ¯2 1 2) ↑ H⍳(M/[2]G)+.×+/[2]R × L
```

whose constituent shapes are:

R	3 4 4
L	3 3 4
G	2 8 5 6 3
M	8

We first bring the expression into standard form by a scan from the right to the left to accumulate ranks.

```
R ← ×/[1](3 4 1 2)⍉+/[2](1 3 ¯2 1 2) ↑ H⍳(M/[2]G)+.× +/[2]R × L
                                        5     5    2    3
3  3            4                     5 5
```

```
R ← ×~/φ+~/ Hι((3↑M)/(2 1 ¯2 1 3)↑(1 5 3 4 2)⍉G)+~.×+/φ 2 3 1⍉R × 2 3 1⍉L

R ← ×/[1](3 4 1 2)⍉+/[2](1 3 ¯2 1 2)↑Hι(M/[2]G)+.×+/[2]R × L

(1)  ×/(4 1 2 3)⍉(3 4 1 2)⍉+...

(2)  ×/(2 3 4 1)⍉+ ...

(3)  ×~/φ(2 3 4 1)⍉+/[2] ...

(4)  ×~/φ(2 3 4 1)⍉+/(1 5 2 3 4)⍉ ...

(5)  ×~/φ+/(2 3 4 1 5)⍉1 5 2 3 4⍉ ...

(6)  ~×/+~/φ[4]φ(2 5 3 4 1)⍉(1 3 ¯2 1 2)↑Hι ...

(7)  ...(φM/[2]G)+{ φ[4]φ(2 5 3 4 1)⍉(1 3 ¯2 1 2)↑1 2 3 4 7; 5 6}~.×φ(2 1)⍉+/[2]R × L

(8)  ...((1 3 ¯2 1 U1)↑φM/[2]G)+{ φ[4]φ(2 5 3 4 1)⍉1 2 3 4 7; 5 6}...

(9)  ...+{ φ[4]φ7 1 3 4 2; 5 6}...

(10) ...((1 3 4 2 5)⍉(1 3 ¯2 1 U1)↑φM/[2]G)+{ φ[4]φ7 1 2 3 4; 5 6}...

(11) ...(φ[3]φ[4](1 3 4 2 5)⍉(1 3 ¯2 1 U1)↑φM/[2]G)

                         +{...

(12) ...                      (1 3 ¯2 1 U1)↑M/[2]φG)...

(13) ...                ((1 3 4 2 5)⍉(3↑M)/[2](1 3 ¯2 1 U1)↑φG)

                 φ[3]φ[4](3↑M)/[3](1 3 4 2 5)⍉...

(14) ...              ((φ3↑M)/[3]φ[3]φ[4](1 3 4 2 5)⍉...)

     ((1 2 4 5 3⍉(φ3↑M)/(1 2 5 3 4)⍉φ[3]φ[4](1 3 4 2 5)⍉...

(15) ...((1 2 4 5 3)⍉(φ3↑M)/φφ[3](1 5 3 2 4)⍉(1 3 ¯2 1 U1)↑φG)

                 +{ 7 1 2 3 4 ; 5 6}(2 U2)↑φ(2 1)⍉ +/[2]R × L

(16)                ...(2 U2)↑φ(2 1)⍉ +/(1 3 2)⍉ R × L

(17)                  ...φ +/(2 3 1)⍉ R × L

                 ...+/(2 U2 U3)↑φ[2](2 3 1)⍉ R × L

(18)             ...+~/(φ(2 U2 U3)↑(2 3 1)⍉φ[1]R)×φ(2 U2 U3)↑(2 3 1)⍉φ[1]L
```

The final expression is:

R ← ×$\tilde{/}$+$\tilde{/}$H⍳((1 2 4 5 3)⍉(ϕ3↑M)/ϕϕ[3](1 5 3 2 4)⍉(1 3 ¯2 1 U_1)↑ϕG)

+{7 1 2 3 4; 5 6}$\tilde{.}$×+$\tilde{/}$(ϕ(2 U_2 U_3)↑(2 3 1)⍉ϕ[1]R)×

ϕ(2 U_2 U_3)↑(2 3 1)⍉ϕ[1]L

where U_1, U_2, U_3 are computed directly from the shapes of the arrays G and R directly. The notation $\tilde{/}$ and $\tilde{.}$ refers to operations using the direct order given by stream production. The following transformations were used on the inner product: Assuming ρρA is 3 and ρρB is 3:

A +.×B <u>is</u>

(ϕA)+{1 2 4 5; 3 6}$\tilde{.}$×ϕ(3 1 2)⍉B

where {1 2 4 5; 3 6} defines the order of delivery of the components. The corresponding stream diagram has already been given. If we have, for example,

(4 2 3 1)⍉A+.×B

it becomes, successively,

(ϕA)+{ (4 2 3 1)⍉1 2 4 5; 3 6}$\tilde{.}$×ϕ(3 1 2)⍉B

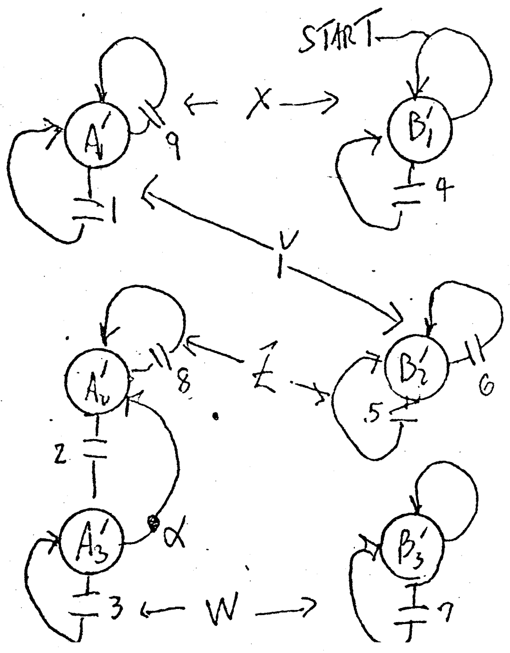
...+{5 2 4 1; 3 6}$\tilde{.}$×...

and, finally,

((2 1 3)⍉ϕA)+{4 1 5 2; 3 6}$\tilde{.}$×(2 1 3)⍉ϕ(3 1 2)⍉B

or A'+{4 1 5 2; 3 6}$\tilde{.}$×B'

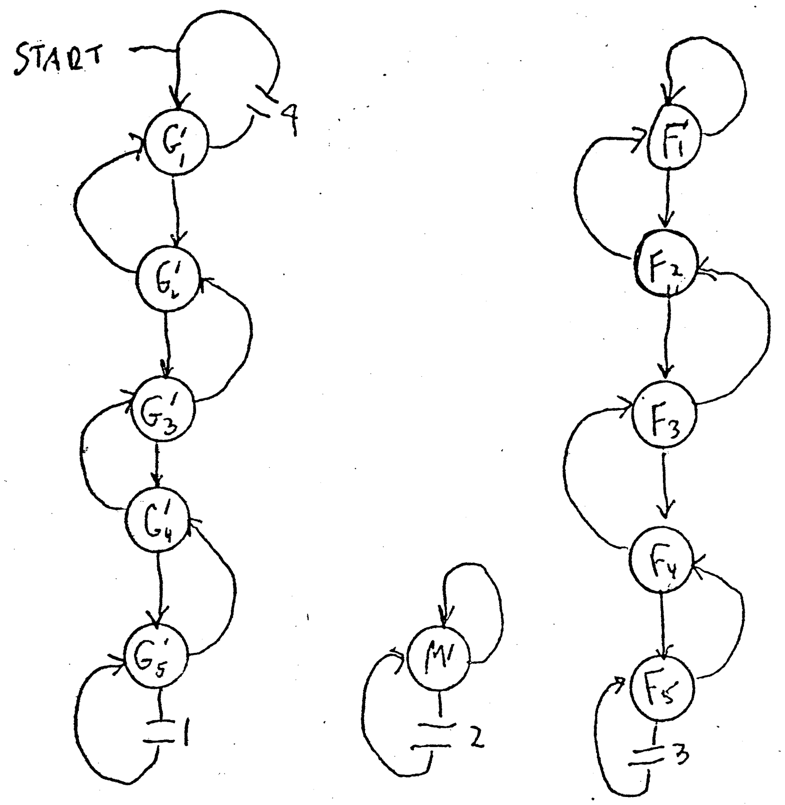
whose stream diagram is:



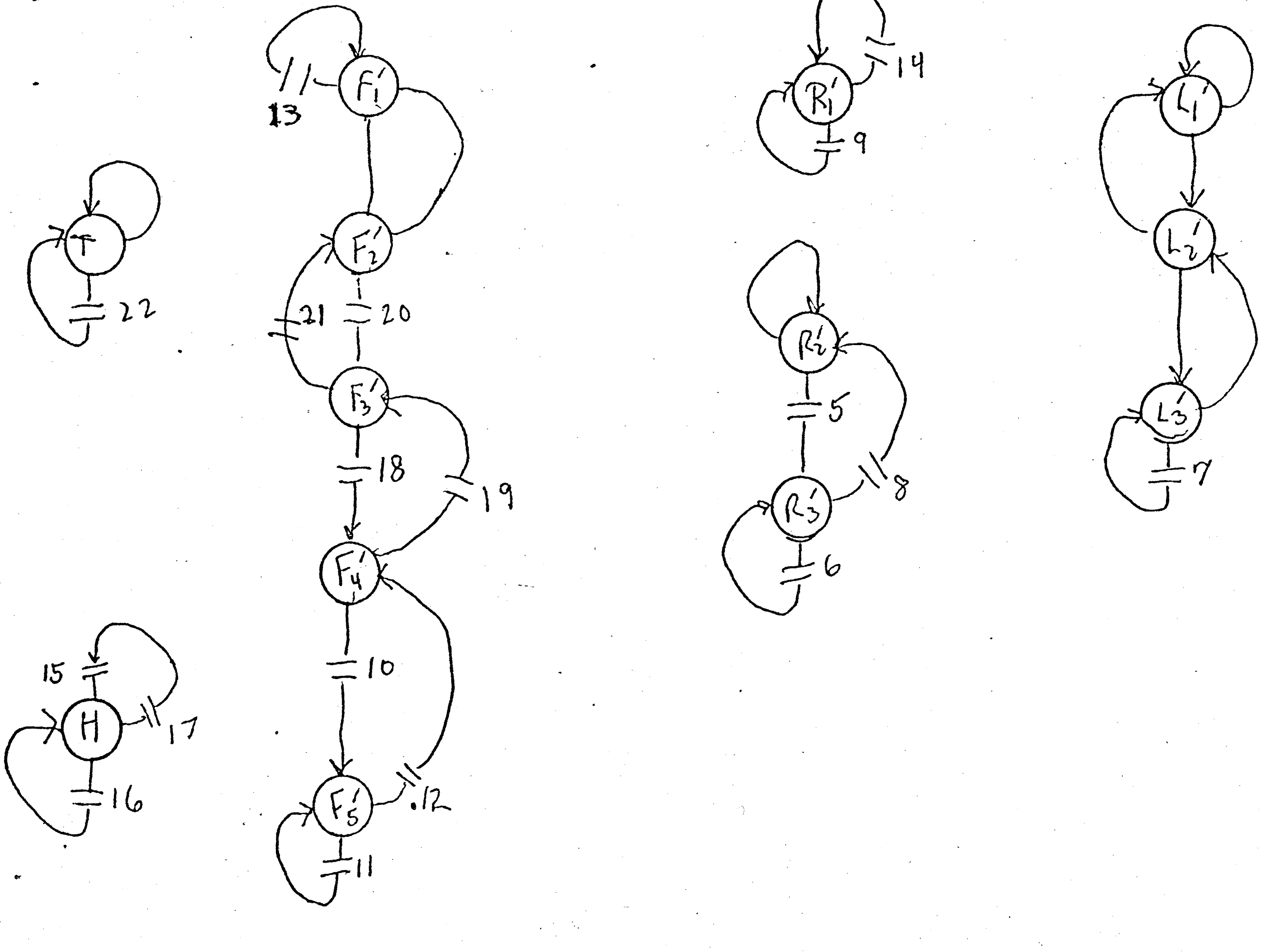
1. ↪Y
2. S ← 0
3. t ← M[$\tilde{\pi}$]; ↪W
4. ↪X
5. ↪Z
6. ↪Y
7. S ← (t×M[$\tilde{\pi}$])+S; ↪W
8. ↪Z
9. ↪X

α. result available as

(a)


START
G_1'
G_2'
G_3'
G_4'
G_5'
4
1
M
2
F_1
F_2
F_3
F_4
F_5
3

(b)


T
22
H
15
17
16
F_1'
13
F_2'
21
20
F_3'
18
19
F_4'
10
F_5'
12
11
R_1'
14
9
R_2'
5
8
R_3'
6
L_1'
L_2'
L_3'
7

The splices refer to the following code:

```
 1.  $t_1 \leftarrow M[\tilde{\pi}]; \hookrightarrow q_1$
 2.  if $M[\tilde{\pi}] = 1$ then $\hookrightarrow q_2; \hookrightarrow q_1$
 3.  $M[\tilde{\pi}] \leftarrow t_1; \hookrightarrow q_2$
 4.  $\hookrightarrow q_3$
 5.  $t_2 \leftarrow 1$
 6.  $t_3 \leftarrow M[\tilde{\pi}]; \hookrightarrow q_4$
 7.  $t_2 \leftarrow (t_3 \times M[\tilde{\pi}]) + t_2$
 8.  $t_4 \leftarrow (t_5 \times t_2) + t_4; \hookrightarrow q_5$
 9.  $\hookrightarrow q_6$
10.  $t_4 \leftarrow 1$
11.  $t_5 \leftarrow M[\tilde{\pi}]; \hookrightarrow q_5$
12.  $\hookrightarrow q_7; \; t_6 \leftarrow r_1 + t_6$
13.  $\hookrightarrow q_6$
14.  $\hookrightarrow q_3$
15.  $k_1 \leftarrow 1; \; r_1 \leftarrow 1 + \rho H$
16.  if $M[\tilde{\pi}] = t_4$ then begin $r_1 \leftarrow k_1$ go to 17 end
     else $k_1 \leftarrow k_1 + 1$
17.  $\hookrightarrow q_7$
18.  $t_6 \leftarrow 0$
19.  $t_7 \leftarrow t_6 \times t_7$
20.  $t_7 \leftarrow 1$
21.  $\hookrightarrow q_8$
22.  $M[\tilde{\pi}] \leftarrow t_7; \hookrightarrow q_8$
```

Initial values for the q_i: (all refer to 1 _of_ the appropriate ladder)

1. M'
2. F' (a)
3. R'_1
4. L'
5. R'_2
6. F' (b)
7. H
8. T

The initialization of all arrays, i.e., affect of transposition, reversal, etc, is omitted. T is the temporary location for R's values until the stream is completed. Then R points to the values held by T. The temporary storage locations are t_1, t_2,...,t_7. The array F is the only array temporary storage.

The output of the parse is the expression in Polish Postfix Form (PPF) or some variant thereof. The compiler proceeds to create "ladder code". What information is needed: Primarily it is the rank of every sub-expression.

When can an APL expression be compiled? And into what? We assert:

(1) Only functions are to be compiled.

(2) Only function lines whose expressions are rank-invariant will be compiled. An expression is rank-invariant iff successive executions find the ranks of all sub-expressions unchanged from those holding during the first execution of the expression.

(3) A function is in a compiled state if any of its lines are compiled.

(4) Initially a function is in the uncompiled state. On first execution the ranks of the actual parameters serve to fix ranks. Any subsequent execution with different ranks converts the function to the uncompiled state--and it remains that way.

(5) What is the class of rank-invariant expressions? Actually almost all APL expressions are rank-invariant. Those which may not be are those containing occurrences of:

(1) function calls

(ii) X ρ ε where X is a computed vector

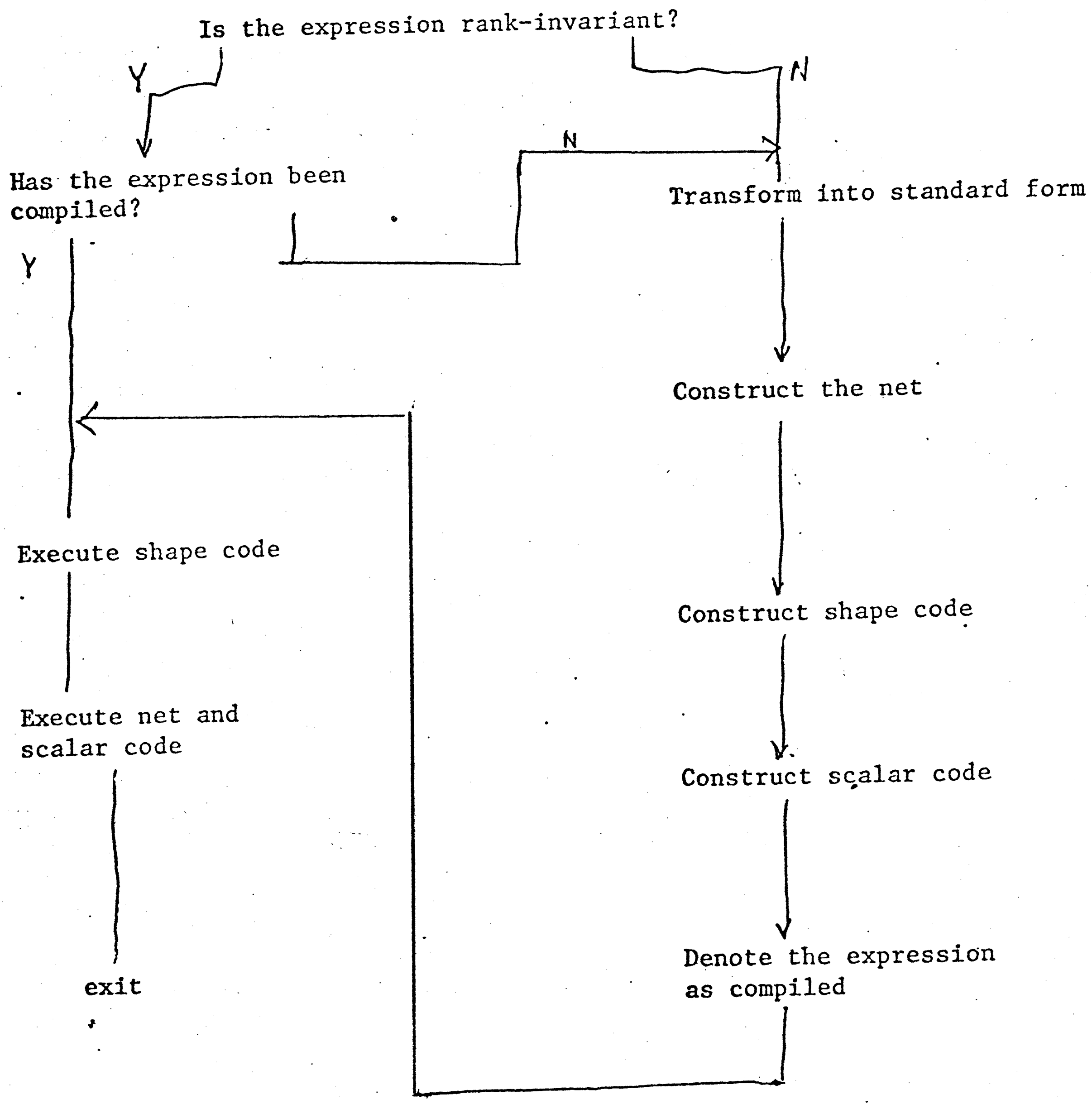
(iii) X ϕ τ where X is a computed vector

Some functions may be easily shown to be rank-invariant (their results are rank-invariant) so that

(1) may be changed to:

(i) Functions calls on functions not known to be rank-invariant.

The flow of control for executing an expression is:


Is the expression rank-invariant?
Y
N
Has the expression been compiled?
N
Y
Transform into standard form
Construct the net
Construct shape code
Construct scalar code
Denote the expression as compiled
Execute shape code
Execute net and scalar code
exit

For those operators where the rank must be computed, the compiler decomposes the expression into disjoint sequences of expressions, disjoint in the sense that one doesn't commence executing until another has terminated. For each operation we need keep note of the 5 connectors (entry, exit result produced, line of result production and communicating register) and the identity of the scalar value(s) transmitted.

A complete example:

H ← R/(D °. = ⍉E)+.×C

1. The expression is rank-invariant. Let the ranks be:

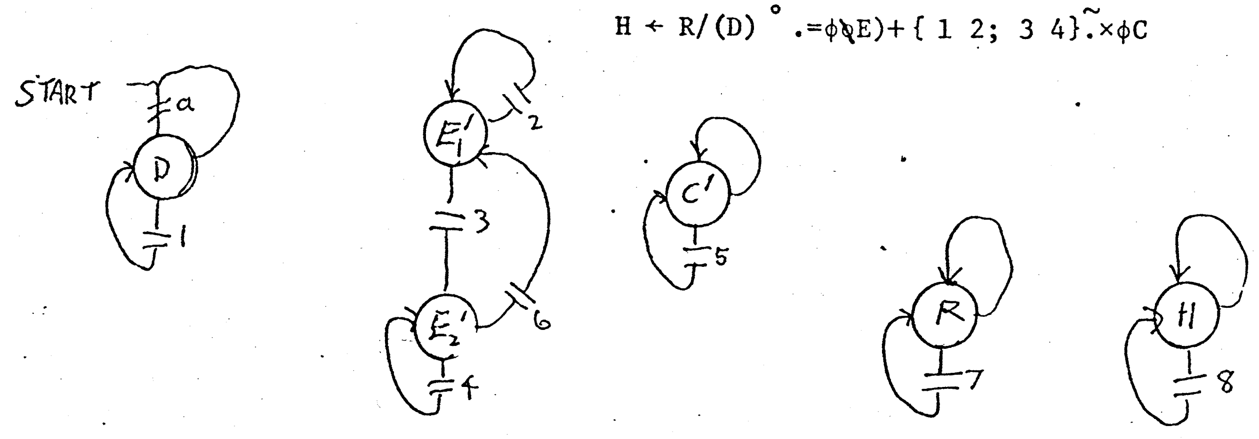
array	rank
C	1
E	2
D	1
R	1

Since H will be filled in ravel order knowledge of its rank is unnecessary. However it may be computed to be:

H	2

2. left to right transformations required yield:

H ← R/(D) °.=ϕ⍉E)+{ 1 2; 3 4}~.×ϕC



the communication registers are: q_1, q_2, q_3, and q_4.

3. The shapes of the arrays are, of course, variable

<u>array</u>	<u>shapes</u>
C	n_1
E	$n_2\ n_3$
D	n_4
R	n_5

4. The code piece establishes the conformability conditions:

a. <u>if</u> $(n_2 \neq n_1) \vee (n_5 \neq n_3)$ <u>then</u> error

$\delta_E \leftarrow$ C DELTR E ; $\delta_E \leftarrow$ REV E ; $\delta_C \leftarrow$ REV C

The other code pieces are:

```
1.  t_1 ← M[π̃];↪q_1
2.  ↪ q_1
3.  t_3 ← 0
4.  t_2 ← t_1 = M[π̃];↪q_2
5.  t_3 ← (t_2 × M[π̃]) + t_3;↪q_2
6.  ↪ q_3
7.  if M[π] = 1 then ↪q_4; ↪ q_3
8.  M[π̃] ← t_3; ↪q_4
```

The function call C DELTR E computes:

$$\begin{pmatrix} 1 & 1 - (\rho_E[⍋ C])[1] \\ 0 & 1 \end{pmatrix} +.\times\ G_E[⍋ C]$$

In this case: C is 2 1. G_E <u>is</u> n_3, 1. REV E computes δ for reversal.

Entry is to the entry point of D. Exit is to the exit point of D.

References

[1] Falkoff, A. D. and K. E. Irverson, "APL/360 User's Manual", IBM Corp., 1968.

[2] Abrams, P. S., "An APL Machine"; Ph.D. Thesis, Stanford University, 1970