Contributed Articles

Iota Flow with Direct Local Functions

Richard H. Oates

0. Introduction

The αω-technique for defining functions which appeared in Iverson's Elementary Analysis [6] has since been called direct definition to distinguish it from the original APL technique called canonical (or del) definition. There are two kinds of direct definition: simple definitions which can be taken as "begin-end" structures, and conditional definitions which are "else-if-then" structures. A program may be used to change the form of a definition from direct to canonical; the function may then be established with a global name and used in the way that functions fixed and established with del are used.

We propose that the del-technique be extended to establish a distinct local function from each internal direct definition in αω-notation, while it also establishes a global function from a definition not in αω-notation. Many branches could then be replaced with bonds of appropriate valence to the new functions in the local domain. Bonds are argument and result linkages; a more impressive word, "call", appears in other languages while the APL word "use" is sometimes not impressive enough.

Local functions with suggestive names can elucidate the structure of a global function in the way that global functions with suggestive names now elucidate the structure of a system. By facilitating such definition, branches would be required much less often, and eventually perhaps not at all--control technique would be unified! In addition, simple local functions could replace "tool" functions (small, broadly useful programs) that are mostly global now. This subordination of function names would make the workspace easier to use and would also facilitate the construction of secure systems.

Some of the new simple local functions can actually be defined globally, and they are so defined in the classroom, but they do not often appear in production workspaces because they fragment the logic of the system, create too many global names, and--all things considered--even the careful programmer frequently finds it easier to take a branch.

Much of the work done by local functions is similar to that done by control structures in other languages, but the argot of control structures does not appear here, since the new technique is just fundamental APL--almost!

The control problem and direct definition are reviewed in Sections 1 and 2, and local definition is introduced in Section 3. Section 4 discusses simple functions and the need for several statements. Section 5 introduces conditional functions, and Section 6 unites conditional and simple functions in the central examples of Iota Flow. Use of system constants as statement labels is proposed in Section 3.1.1 as a related topic.

1. Flow

It has been said that potent language primitives make additional control unnecessary in APL, but technical advance always whets the appetite. A system will grow until the quantity of its explicit control matches the capacity of its explicit controllers. What changes is the amount of work that gets done, and the principle is as old as the development of the wheel or the hand. The phrase "potent language primitives" brings to mind monadic iota (how unconscious now is use of this function which first astonished me) or perhaps a smooth and-scan on a rank-3 transposition (it does not, alas, suggest the management of non-trivial conversation).

Control today requires much branching, so it is likely that branching will persist long after more potent primitives are developed. One might wish to have, for example, a function that tests its input against an exhaustive table of valid non-numeric responses in the way that quad input guarantees a numeric response.

The branch in APL is disconcerting--there are too many labels and too many branches in my programs! The effect of a right arrow is determined by a condition through various strategies yielding two opposite meanings. We have one statement that branches on true or continues on false,

another statement that branches on false or continues on true, and programmers casually use a variety of expressions to produce whichever statement may save a cycle or two. Branching has been much discussed since Dijkstra [2] touched the nerve in 1968. All control injects discontinuity, and what the branch disconnects is too often arbitrary. What indented languages always disconnect is every "continuity" except the innermost in every control nest (only the deepest structure is not interrupted by a deeper structure). With its end firmly fastened to its beginning, the defined function has a much stronger form. What the function disconnects is detail, the explanation (definition) of its name which may well include the names of other defined functions. Some idea of how any system works can be gained merely from a graph of the use of its defined functions. If the system is well written (see Appendix A), this idea can be clear indeed.

The control technique of APL is the marshalling of functions, defined and primitive alike, from right to left across the page except for operators and parentheses. I have been an application programmer in the same department since 1957, writing large and nasty programs for large and nasty files in every languge back to IBM-650 Soap. I remember the school day when we were taught the IBM-1401 SBR instruction that made "closed routines" so much easier to write, and I remember the delight of discovering in Cobol that you did not always have to "perform"--if you left the periods off a group of statements in an if-nest they would execute as a whole. I have used them all, heavily, and I believe that no other structure is half as good for the management of control as the APL defined function--there are not enough functions in my programs!

2. Direct Definition

Iverson's two forms of direct definition are:

```
simple:val
conditional:val:prop:val
```

The name of the function appears to the left of the colon and the syntax and localizations are derived from the value and proposition expressions. There is always an explicit result; α is taken as the first argument and ω as the last; variables set within are local; global names may be referenced. The function name itself is always global. If the proposition is false (0) the left value expression is executed and if true (1) the right is chosen. Branching is not allowed. A defined function *DEF* (see Appendix B) can change an input direct definition to canonical form and then fix the function in the workspace. In the following sample execution of *DEF*, the whole second line (input) defines the function *AVG* and the third line is the name explicitly returned by *DEF*:

```
      DEF
AVG:(+/⍵)÷⍴,⍵:0=⍴,⍵:'EMPTY'
AVG
      AVG ⍳0
EMPTY
```

Since direct definition was invented for the pursuit of mathematics, in Iverson's examples the first expression often contains a recursive reference. I shall be discussing direct definition in the pursuit of programming where recursion occurs much less often. Iverson uses one direct function within another at several points in his Turing Lecture [9]--particularly *CFD* and *RFD* in the discussion of representations--but all are global in a workspace that must contain more than fifty functions, most of which are not of interest in any given section of the lecture. Some of these could be local, and in the usual workspace focused on a single topic, many more could be. Other languages allow procedures to be defined within procedures to any depth. A merely two-level definition technique in APL would provide automatic localization of the second-level functions, unlocking a whole new domain over which the net of automatic function bonding could be drawn. Examples of the need for simple local function bonding can usually be found in the nearest program. In the *DEF* "system", the main function is *R*9 and it uses *I*9 and *F*9 at several places in its text. If these two functions could be made local to *F*9 as easily as the variables *A*9 and *C*9 are, then *DEF* would be more convenient for its users, because its "system" would be less obtrusive, but no less convenient for its author and its auditors. An inconvenient technique for localizing *I*9 and *R*9 today appears in Appendix C which describes the program *FI*.

3. Local Definition

The modification of direct definition described below is intended for both local and global use, but the latter is not pursued in this paper except in Section 3.4. Local definition is an extension and permutation of direct definition.

3.1 Syntax. In a local definition there is no need to limit the number of value expressions to one or two, particularly if a vertical arrangement (one expression per displayed line) is allowed as an alternative to the horizontal. The proposition (fork) may then precede the first value and become integer valued:

```
IOTA:FORK
:VAL
:VAL
:VAL
...
```

The definition terminates at the first line that does not start with a colon. Only the name of the function and the first value are required.

If the fork is missing, the function is simple and the statements are executed sequentially from the left or top. Simple functions arranged vertically could contain branches (to "system constants" as described in Section 3.1.1). When the fork is present, one expression selected by the fork is executed, as described in Section 3.5. (This expression may not be a branch.)

3.1.1 System constants. The colon that terminates the name of a local function conflicts with the colon that terminates a statement label. Labels are a flawed form of descriptor, as can be seen in this example, where *L3* is unwanted and *L1* need not appear:

```
→CONDITION/L2
(FIRST CASE)
→L3
L2:(SECOND CASE)
L3:...
```

In a new function, usually I just take labels from the the series *B*, *D*, *F*, *H*, etc. A label is a "local constant" which generates a syntax error if it appears immediately to the left of a specification arrow, but it does not look like a constant. In [4] uniformity is described as "rules are few and simple" and generality as "a small number of general functions provide as special cases a host of more specialized functions." These principles might be better served if statement labels were local system constants with implicit access control (assignments ignored). Names like ⎕3 and ⎕14.2 could be automatically reconstructed when the user leaves definition mode (each occurrence of the name ⎕14.2 would be changed to, perhaps, ⎕15). Because a system constant is a name, the programmer would not have to revise branch destinations after insertions. Because they are also numbers (prefixed by a quad), they would not need to appear at the left end of the statement. In a local definition their entry could be forbidden; in a global definition they could be entered for emphasis, but they would not be exdented in display. System constants would be more convenient and less obtrusive than statement labels and they would allow the colon to be used unambiguously both ways. An example appears in Appendix D. The system variable ⎕*LC* would take the value of the current system constant, explicit or implied.

In this paper, the colon is used ambiguously, and most labels are taken from the series *L1*, *L2*, *L3*,... .

3.2 Recursive definition. A direct-definition value may not be a new local definition; two levels of definition are sufficient for practical purposes. For example, if the definition of a function named *ABLE* has three values, the second cannot be exploded into a whole new definition of a function called *BAKER*. *BAKER* can easly be defined after *ABLE* and should not need to be defined within *ABLE*. Recursive definition would complicate the syntax with no apparent advantage.

3.3 Explicit result. In addition to α and ω, a third symbol may be used to identify a result not generated at the conclusion of the last or only expression. For convenience the result symbol used in this paper is $\subset$, although a character that my terminal will not form (ω backspace _) might be a better choice.

3.4 Scope. If del definition could be entered with an empty header line, each local definition would then become a distinct global function, since no global name is provided to reduce its scope. For example:

```
∇
  DIV::(ω≠0)×α÷ω+ω=0
∇
```

If the header line is not empty, then the local definitions become local functions.

3.5 Use. The name of a local function has ordinary local scope and the function can be used wherever its name is known. Although global and local definitions share a common set of statement numbers, global-function execution skips over the local definitions as if they were not there. If a branch in any function evaluates to a number not assigned to a statement in its own definition section, the function returns.

If a fork evaluates to a number outside its active range, its function returns. The range is determined in one of two ways. If there is only one subsequent value, the function works as a conditional-execution statement, so its range is restricted to 1. If there are two or more value expressions, the first expression is executed when the fork evaluates to 0, the second when it evaluates to 1, etc. These forms are:

```
IOTA: FORK : 1-VAL
IOTA: FORK : 0-VAL : 1-VAL : 2-VAL : ...
```

4. Simple Functions

The functions *I9* and *R9* discussed in Section 2 can now be easily localized in *F9*. In this example the function *I9* is defined in the first statement after the header and used initially in the fourth.

The missing fork (between the two colons) means that the function is simple:

```
D←F9 E;F;I;J;K;Q;□IO
I9::(α∘.,=ω)∧((ρω),ρω)ρ~2|+\ω=''''
R9::(,((1↑α)I9ω)∘.,≠N↑1)/,ω,((ρω),¯1+N←ρα)
  ρ1↓α
D←(2ρ□IO←0)ρ''
→((2|+/E='''')∨∧/ 1 3 ≠+/':' I9 E)/ρD
...
```

*I*9 and *R*9 are examples of a large class of "tool" programs that are specialized but needed more than once, well defined (stable), and small, but not necessarily of one line. (Widely used "tools" are TSIO's *TRY* and *CHK*, which open a file and check the return code after reading a block. Appendix D shows αω-versions of *TRY* and *CHK*.)

The function *DEF* works by substituting the four characters ' *X*9 ' for 'ω', and introduces *Z*9 as the explicit result. See the text it compiles for *AVG* in Appendix B. *FI*, the local-definition version of *DEF*, employs the same technique, although it does not work properly on Statements 7 of *TRY* and 5 of *CHK*, where ω and α appear in quotes. *FI* merely changes the form of a definition from direct to canonical and then fixes it. If the function *FI* modeled more of the APL interpreter, the problem would go away and *X*9, *Y*9, and *Z*9 would also be replaced by internal temporaries fully isolated from names chosen by the user. For the purpose of this paper, *FI* makes substitutions for α, ω, and ⊂ regardless of quotes.

If locked function *F* accesses secure data and then uses global function *G*, an intruder can substitute his own version of *G* and use it to "see" the local variables of *F*. In a secure system all routines needed by *F* must by locally coded in *F* or locally fixed in *F* after being read from a file. Secure systems are frequently file-management systems, and a file system used for information retrieval may have hundreds of functions and dozens of users, some of whom write their own APL functions. In systems like these, the management of names is not a "secondary" problem [7].

5. Forking Functions

N-way branches are coded in APL today in two unattractive ways. If each action can be coded on a single line and the routine is stable, then addition can be performed on the label constant. In this example, *I* is expected to have the value 1, 2, or 3:

```
...
L1:→L1+I
→L2,0ρ(FIRST CASE)
→L2,0ρ(SECOND CASE)
(THIRD CASE)
L2:...
```

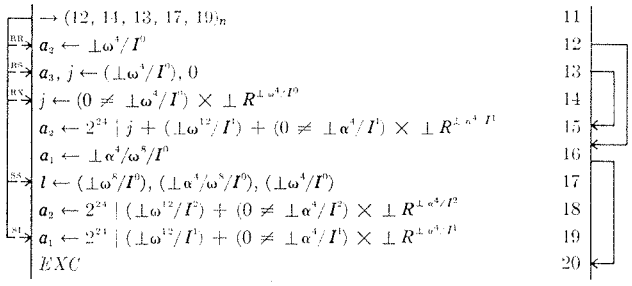
A forking function does the same thing without labels. An illustrative example is the quadratic formula, "minus *B* plus or minus the square root of *B* squared minus four *A C*, over two *A*". This formula has three types of result, and it can be coded in five lines, with the second line taking care of the case when *A* is 0:

```
...
Z←QUAD ¯1 100 ¯2000
QUAD:(ω[0]≠0)×2+×D←(ω[1]*2)-4××/ 1 0 1 /ω
:'ZERO DIVIDE'
:'COMPLEX'
:2ρ(-ω[1])÷2×ω[0]
:((-ω[1])+R,-R←D*.5)÷2×ω[0]
...
```

If some paths from the fork are too long to fit on a single line, then today a vector of labels provides more flexibility: →(*L*1,*L*2,*L*3,...)[*I*]. In a situation like this, a fork can work together with other (local and global) functions to provide as much flexibility as the branch statement but in a clearer way. An example appears in the next section.

6. Iota Flow

The IBM-360 effective-address calculation of [3] provides an example of the way in which local functions can work together.



In this document, which appeared in 1964 before the computer implementations of APL, α/ and ω/ are used as ↑ is today; the expression following ⊥ in Line 16 means: take the first four of the last eight elements of Row 0 of *I*. The variable *I* is a (3 16)-bit instruction register; *R* is sixteen general registers; *N* determines the instruction type; one, two, or three addresses are left in *A*; and for an SS-instruction, three lengths are left in *L*. A literal rendering of this algorithm in current APL is:

```
...
→(RR,RX,RS,SS,SI)[N]
RR:→L2,A[2]←2⊥¯4↑I[0;]
RS:→L1,A[3]←2⊥¯4↑I[J←0;]
RX:J←(0≠2⊥¯4↑I[0;])×2⊥R[2⊥¯4↑I[0;];]
L1:A[2]←(2*24)|J+(2⊥¯12↑I[1;])+
  (0≠2⊥4↑I[1;])×2⊥R[2⊥4↑I[1;];]
L2:→L3,A[1]←2⊥I[0;8+⍳4)
SS:L←(2⊥I[0;8+⍳8]),(2⊥I[0;8+⍳4]),
  2⊥I[0;12+⍳4]
A[2]←(2*24)|(2⊥¯12↑I[2;])+(0≠2⊥4↑I[2;])
  ×2⊥R[2⊥4↑I[2;];]
SI:A[1]←(2*24)|(2⊥¯12↑I[1;])+(0≠2⊥4↑
  I[1;])×2⊥R[2⊥4↑I[1;];]
L3:EXC
...
```

Multiple specification was not yet in the language. This carefully constructed algorithm is a classic small example of branch optimization on irregular routines, which is a tedious chore.

Local functions eliminate all branches and reduce the count of characters to approximately what it was in 1964:

```
...
AL[1+⍳ 2 2 3 6 1[N]]←ADRS N
⍝ RR RX RS SS SI
ADRS:⍵
:R 1 2
:(R 1),M I+(B 1)+D 1
:(R 1),(M(B 1)+D 1),R 2
:(M(B 1 2)+D 1 2),0,L
:M(B 1)+D 1
⍝ MOD,REGISTER,INDEX,BASE,DISP,LENGTH:
M::(2*24)|⍵
R::2⊥⍉I[0;8 12[⍵-1]∘.+⍳4]
I::(0≠R 2)×2⊥R[R 2;]
B::(0≠2⊥⍉I[⍵;⍳4])×2⊥⍉R[2⊥⍉I[⍵;⍳4];]
D::2⊥⍉I[⍵;4+⍳12]
L::(16⊥R 1 2),R 1 2
...
```

The fan-out branches become a five-way fork (*ADRS*) leading to five expressions, each of which is confined to a single line by the more supple use of functions that local definition permits. The fan-in branches simply disappear. The functions *M*, *R*, *I*, *B*, *D*, and *L* would never be defined as global functions because that would disperse the algorithm, so branches are usually coded. On the other hand, if local functions were even simpler than branches, they would probably be used. This example illustrates the way in which local functions can elucidate program structure. The algorithm appears in five statements in a language close to the natural "index plus base plus displacement", using a vocabulary of six new words (functions) which are immediately explained (defined). In order to make this vocabulary complete, two of these functions take no arguments. Because the 1964 program operates on a single level, it cannot be as clear as the 1981 version which operates on three.

Each of the six simple functions contains a single expression, but in some other situation each might well contain several expressions in simple or fork form. Other languages allow one program to be defined inside another, but the technique is more promising in APL because a local function can delimit a group of statements as readily as it enumerates a set of alternatives, and because the simple valence mechanism of APL makes it easy to bond one function to another. (I once lost two weeks using a batch language trying to pass the kind of argument--numeric or character and scalar or vector or matrix--that I now take for granted; six months later I was told that what I had been trying then had not been working.) Iota flow is automatic bonding in a simple form that groups consecutive or alternative statements. Iota flow makes an arbitrary network of related conditions easy to construct to any dynamic depth, even though the number of static levels of function definition is limited to two. The superficial appeal of if-then-else quickly fades with deeply nested structures, where it becomes apparent that the syntax is the problem. The forking function is simpler and more general than if-then-else, which it encompasses without special provision.

These remarks also apply to niladic functions (in this example *I* and *L*) which in some situations might bond to other functions to complete their work. For this reason, and because a niladic function is useful as the initial function in a (possibly conversational) system, I would not like to see them dropped [8]. (I would also like to see ambivalence extended to zero valence. A printing system, for example, might have several options, *X*, *Y*, and *Z*, with *X* used most often. Since the absence of *X* can imply *X*, a simple *PRINT* statement can be introduced early in a teaching situation with the full non-default range--*PRINT X*, *PRINT Y*, or *PRINT Z*--held for later.)

In the preceding example, local functions were used to articulate an algorithm that appeared quite early in the development of the language; in the next example they consolidate a more recent, fully articulated algorithm without losing any of the pieces. The compiler diagrammed in Appendix A contains one branch statement in each of four functions, *PARSE*, *STRIP*, *POLISH*, and *COMPILE*, and these functions together with *CENTER* are exhibited here to facilitate comparison. The index origin of the compiler is 1:

```
Z←PARSE E
→0×⍳∧/FUNCTIONS Z←STRIP E
Z←(' ',' ',PARSE L Z) ON(C Z) ON ' ',' ',
  PARSE R Z

Z←STRIP E
→0×⍳1≠⌊/DEPTH Z←E
Z←STRIP 1↓¯1↓E

Z←POLISH M
Z←CT M←(∨/[1]M≠' ')/M
→0×⍳1≥1↑⍴M
Z←Z,(POLISH LT M),POLISH RT M

Z←COMPILE E;CE
CE←CENTER E
Z←(((('←'∊CE)⍱3≥⍴E)/NAMES[1],'←'),
  CE[2 1 3]
NAMES←1⌽NAMES
→0×⍳3≥⍴E
Z←Z ON COMPILE(LEFT E),Z[1],RIGHT E

Z←CENTER E
Z←(LOCCENTER E)/E
```

Each of the nineteen global programs in the original compiler becomes a local

function in the version *DLF* below. The fourteen programs that do not appear above were converted merely by changing their "punctuation". For *PARSE* and *STRIP* the change was almost as simple: complementing the condition. Alternatively *STRIP* might be coded as a conditional execution, but the presence of quotes make this treatment of *PARSE* inconvenient. Since the relation is not conspicuous, it should perhaps be mentioned that *STRIP* is a forking function used within the fork of *PARSE*. In *POLISH* and *COMPILE*, the elevation of the condition to the top of the program produces a function with a sharper balance and lower voltage (the fork definition seems less "algorithmic" than the branch statement). In order to achieve this balance in *COMPILE*, part of the specification is transferred to the *CENTER* function:

```
P←DLF S;⎕IO;N
⎕IO←0
P←COMPILE POLISH PARSE S
⍝
PARSE:∨/FUNCTIONS⊂←STRIPω
:(' ',' ',PARSE L⊂) ON(C⊂) ON ' ',' ',
  PARSE R⊂
C::ω[CENTRALFNω]
L::(¯1+CENTRALFNω)↑ω
R::(CENTRALFNω)↓ω
CENTRALFN::((FUNCTIONSω)∧0=DEPTHω)⍳1
STRIP:1=⌊/DEPTH⊂←ω:STRIP 1↓¯1↑ω
DEPTH::+\(ω='(')-0,¯1↑ω=')'
⍝
POLISH:1<1↑ρω←(∨≠ω≠' ')/ω
:CTω
:(CTω),(POLISH LTω),POLISH RTω
CT::, 1 1 ↑(' '≠FIRSTCOLω)/ω
RT::(∨\¯1⌽' '≠FIRSTCOLω)/ω
LT::(~∨\' '≠FIRSTCOLω)/ω
FIRSTCOL::,((1↑ρω),1)↑ω
⍝
COMPILE:3<ρω
:CENTERω
:⊂ON COMPILE(LEFTω),(1↑⊂←CENTERω),RIGHTω
CENTER:
:CE←(LOCCENTERω)/ω
:N←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'[1+26|ρ
  ⎕LC]
:(((⊂'←')∈CE)↑3≥ρω)/N,'←'),CE[2 1 3]
LEFT::(~∨\LOCCENTERω)/ω
RIGHT::⊂←(~⊂∨∧\~⊂←LOCCENTERω)/ω
LOCCENTER:
:(⍳ρω)∈ 0 1 2 +(FUNCTIONSω)⌈:×⍳ρω
⍝
ON:
:α←(¯2↑ 1 1 ,ρα)ρα
:ω←(¯2↑ 1 1 ,ρω)ρω
:(((ρα)⌈ 0 1 ×ρω)↑α),[1]((ρω)⌈ 0 1 ×ρα)↑ω
FUNCTIONS:
:ω∈'←+-×÷<≤=≥>≠∨∧?∈ρ~↑↓⍳○*⍟⌈⌊⊥⊤|'
```

Program listings for most block-structured languages are indented. They attempt to modulate a detailed listing with the profile of its structure. Each format suffers from the other--the detail constantly disturbs the structure, and the structure interrupts the detail whenever control descends to a lower level. In *DLF* the depth of function nesting reaches 5 at two points. This is easily seen in Appendix A, but hard to pick out of the definition. Does it matter? It does not much help the auditor of *DLF* to know that *CENTRALFN* and *FIRSTCOL* are both at Level 4, but it is important to know that *COMPILE*, *POLISH*, and *PARSE* are all at the same level and that it is one step down to *FIRSTCOL* from *CT*, *RT*, and *LT*. Each bond between neighboring functions can be seen clearly in *DLF*. It shows the statements and the "micro-structure" while Appendix A summarizes the structure as a whole. The distinction between the two figures is appropriate, and large. APL is beyond indented blocks. What would be useful is a system variable that could display all function bonding performed in the workspace since the variable was last set to null by the user. Graphs like the one in Appendix A for both trees and forests are easily generated from an $(N,2)$-table where N is the number of unique function bonds, but this table cannot in general be established at the application level, because of ⍎, the self-referential primitive.

In this compiler the functions average 1.5 statements in length. Can you imagine the 778 statements of the formal description of System/360 defined in 520 global functions? Excluding the two literals that begin '*ABC*' and '←+-', the *DLF* program has a total of 792 characters, of which 46% are '*ABCDEFGHIJKLMNOPQRSTUVWXYZ*⎕', 6% are '¯0123456789', 24% are '[(:'';)]', 8% are 'αωC', and only 16% are the symbols '←<=≥≠∨∧+-×∈ρ~↑⍳○⌈⌊|,./⌿\' for which APL is famous. Local functions can change the face of the language.

7. Conclusion

After the primitive operator and the primitive function, the defined function is the best "control structure" of all (except possibly for the defined operator, which I have not yet used). If alternative courses of action are represented in a homogeneous way, then the potent language primitives can do the job, but when the representation or the algorithm is diverse, the branch comes into play. The conditional-direct function and the conditional-execution statement are two special cases of a strong general form, N-way branching, that requires too many labels and too many branches in APL today. The fork is more concise than the conditional branch and it eliminates the unconditional branch that is otherwise required to reunite the flow. It is somewhat less flexible than the branch, but this may be a virtue if it encourages better program structure. One new level of definition is enough; from the global level and one local level, the automatic function bond easily builds networks to any depth.

The prohibition on side effects will limit the use of local definition, particularly until general arrays make

explicit results more general, but it does not sound like a bad idea to complement the license of global definition with more rigorous local definition that can be entered with less clerical effort. Definition on-the-fly is routine in a natural language, and definition within definition can make this most fluent programming language more fluent still. If functions are made easier to write, more will be written and style will improve.

Richard H. Oates
IBM World Trade Americas/Far East Corp.
Town of Mount Pleasant, Route 9
North Tarrytown, New York
USA 10591

References

[1] J.A. Brown. Evaluating Extensions to APL, APL79 Conference Proceedings, APL Quote Quad 9 4 (June 1979) pp. 148-55.

[2] E.W. Dijkstra. Goto statement considered harmful, Letter to the editor, Comm. ACM 11 3 (March 1968). In 1959 I discussed Goto with a psychiatrist, complaining that no one else in the office liked to write a program that can be read straight down the page.

[3] A.D. Falkoff, K.E. Iverson, and E.H. Sussenguth. A formal description of System/360, IBM Systems Journal 3 3 (1964).

[4] A.D. Falkoff and K.E. Iverson. The design of APL, IBM Journal of Research and Development 17 4 (July 1973).

[5] K.E. Iverson. APL in Exposition, APL Press, Pleasantville, New York (1976).

[6] K.E. Iverson. Elementary Analysis, APL Press, Pleasantville, New York (1976).

[7] K.E. Iverson. Programming style in APL, An APL Users Meeting, I.P. Sharp Associates, Toronto (1978).

[8] K.E. Iverson. The role of operators in APL, APL79 Conference Proceedings, APL Quote Quad 9 4 (June 1979) pp. 128-33.

[9] K.E. Iverson. Notation as a tool of thought, Comm. ACM 23 8, (Aug. 1980) pp. 444-65.

Appendix A. Bonding Tree

This is a function-use graph for the compiler in [5]. Vertical paths are traced by the three symbols + ∘ - and horizontal paths by the 26 symbols A-Z. The symbol + at an intersection means that the function name to the right appears once in the tree. Functions that appear more than once are marked with ∘ at the first occurrence and with - at each reoccurrence, whereupon tracing of the network terminates:

```
X ∘ PARSE - PARSE
  +       +
  +       ∘ STRIP - STRIP
  +       +       ∘ DEPTH
  +       +
  +       + L ∘ CENTRALFN ∘ FUNCTIONS
  +       +                 - DEPTH
  +       +
  +       + C - CENTRALFN
  +       +
  +       - FUNCTIONS
  +       ∘ ON
  +
  ∘ POLISH - POLISH
  +        +
  +        + CT ∘ FIRSTCOL
  +        +
  +        + LT - FIRSTCOL
  +        +
  +        + RT - FIRSTCOL
  +
  ∘ COMPILE - COMPILE
            +
            + CENTER ∘ LOCCENTER -
                                   FUNCTIONS
            +
            + LEFT - LOCCENTER
            +
            + RIGHT - LOCCENTER
            +
            - ON
```

The graph is easily limited to the set of all paths that contain a designated function, such as *ON*:

```
X + PARSE ∘ ON
  +
  + COMPILE - ON
```

This compiler, which does not handle names wider than one character because it was intended for tutorial use, was incorporated into a production query system at Americas/Far East Headquarters several years ago. A fifteen-minute change to a few statements let it operate on numeric vectors in place of character strings, each number in the vector being decoded from a name (token) in the string. The maximum width of the token depends on the size of the decoding alphabet and the representation chosen for numbers. The decoding is *(⍴ALPHABET)⊥ALPHABET⍳TOKEN*.

The thirty statements of the compiler appear in nineteen functions, and the nineteen functions appeared in a workspace that contained over 100 functions of its own at that time. The compiler was small enough to use but too visible, so after a few months of operation it was sadly butchered.

The compiler appears as a set of local functions in Section 6.

Appendix B. Direct-Definition Compiler

The *DEF* function is reprinted from [9] with some of its multiple specification rewound:

```
Z9←DEF
Z9←⎕FX F9 ⍞

D←F9 E;F;I;J;K;Q;⎕IO
D←(2ρ⎕IO←0)ρ''
→((2|+/E='''')∨∧/ 1 3 ≠+/':' I9 E)/0
E←, 1 1 ↓⎕CR ⎕FX 'Q',' ',[¯0.5],E
I←':' I9 F←'α X9 ' R9 'ω Y9 ' R9 E
D←(0,-6-+/I)↓(-(3×I)++\I)⌽⍵(7,ρF)ρ(7×ρF)↑F
I←2+ι¯2+F←1↓ρD
D←3⌽(C9[((2⌊2⊥∨/'αω' I9 E),1+I),5;]),⍵D[;0,
  I,1]
J←>≠ 0 ¯1 ⌽'←⎕' I9 E
J←((¯1⌽I)∧J)/K←+\I<0,¯1↓I←E∈A9
K←∨/((-K)⌽I∘.>ι1+⌈/K)[;J-1]
D←D,(F,ρE)↑⍵ 0 ¯2 ↓(K+2×K<1⌽K)⌽' ',E,[0.5]
  ';'

Z←X R9 Y;N
Z←(,((1↑X) I9 Y)∘.≠N↑1)/,Y,((ρY),¯1+N←ρX)ρ
  1↓X

Z←A I9 B
Z←(A∘.=B)∧((ρA),ρB)ρ~2|+\B='''''

      C9
   Z9←
 X9Z9←
 Y9Z9←X9
)/3→(0=1↑,
   →0,0ρZ9←
   Z9←

      A9
012345678
9ABCDEFGH
IJKLMNOPQ
RSTUVWXYZ
ABCDEFGHI
JKLMNOPQR
STUVWXYZ⎕
```

The function resulting from the example in the text is:

```
Z9←AVG Y9
→(0=1↑,0=ρ,Y9)/3
→0,0ρZ9←'EMPTY'
Z9←(+/Y9)÷ρ,Y9
```

Appendix C. Local-Definition Compiler

The function *FI* (Flow Iota), is an adaptation of *DEF* for local functions. *A9*, *C9*, *I9*, and *R9* have become *A*, *C*, *I*, and *R*, and they all are made local to *F9* in order to demonstrate the current awkwardness of localizing function names. The final *⎕FX* operation is left outside *F9* so that it may even redefine a name local to *F9*. *F9* will accept a matrix of expressions, but no further attempt is made to model behavior of a global function. Definitions in αω-form are compiled both for simple arguments and for forks with any number of expressions. The function *I* has been modified to accept α, ω, and ⊂ in quotes as discussed in Section 4.

```
Z9←FI
Z9←⎕FX F9 ⍞

D←F9 E;⎕IO;A;C;E;I;N;R;B;F;I;J;K;M;N;Q;X;Z
Z←⎕FX(D←CR 'F9')[6 7 +⎕IO←0;]
Z←⎕FX D[8 9 ;]
Z←⎕FX D[10 11 ;]
Z←⎕FX D[12+ι3;]
→L1
Z←A I B
Z←A∘.=B
Z←X R Y;N
Z←(,((1↑X)I Y)∘.,≠N↑1)/,Y,((ρY),¯1+N←ρX)ρ1↓X
Z←E B;Q
Z←(Q∨1⌽Q><\Q←B≠' ')/B←,' ',B
Z←N B;Q
B←(Q∧~¯1⌽Q←B[;0]≠' ')≠B←(ιρ⊥)⌽(2ρρB)ρB
Z←(∨/Q∧⍵Q←<\B∧.=⍵B)≠B←(ρB)ρQ\(Q←,∧\B≠' ')/,
  B
L1:C← 0 22 ρA←'0123456789ABCDEFGHIJKLMNOPQR
  STUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ⎕'
→((2|+/''''=E)∨2>+/':' I E←E)/ρD←(2ρ0)ρ''
E←, 1 1 ↓⎕CR ⎕FX 'Q',' ',[¯0.5],E
F←'⊂ Z9 ' R 'α X9 ' R 'ω Y9 ' R E
Z←1+2×M←+/I←':' I F
D←(0,-(Z-1)-M)↓(¯(I×M)++\I)⌽⍵(Z,ρF)ρ(Z×ρF)↑
  F
C←C,[0] '    ;W9Z9←                '
C←C,[0] ' Y9;W9Z9←                 '
C←C,[0] ' Y9;W9Z9←X9               '
C←C,[0] '    )+1→(W9≠1)×W9←1+1↑,('
C←C,[0] '       →0,ρZ9←            '
C←C,[0] '       Z9←                '
⍝(~B←D[;1]∨,≠' ')/'C[ι3;2+ι3]←C[3 4;]←'' ''
  '
⍝(2=F+.=':')/'C[3;3 4]←'' '''
⍝(~B∨~'⊂'∈E)/'C[5;]←'' '''
I←((F←1↓ρD)-3)ρ4
D←5⌽C[(2⌊2⊥∨/'αω' I E),3,I,5;],⍵D
D←(1,B,(¯2+1↑ρD)ρ1)≠D
K←+\I<0,¯1↓I←E∈A
J←((¯1⌽I)∧J←>≠ 0 ¯1 ⌽'←⎕' I E)/K
K←∨/((-K)⌽I∘ ,>ι1+⌈/K)[;J-1]
Z←, 0 ¯1 ↓K⌽' ',[0.5] E
D←D,((F←~B),ρZ)↑(1,ρZ)ρZ←,';',N ' ',E Z
```

Appendix D. Examples

Here are αω-versions of *TRY* and *CHK* in a "quilt" ("cover") function. Quilts establish an aggregate required by current file technique, and serve as a base for a local *⎕FX*. The quilt is gone at run time, so in this exhibit *TRY* and *CHK* are shown branching to their "own" statement numbers; if they were defined as local functions in the sense described in this paper, each system constant would reflect the new position of the local-definition line in the global definition. Statement 7 of *TRY* is discussed in Section 4.

```
Z9←AVG Y9;W9
→(W9≠1)×W9←1+1↑,(0=ρ,Y9)+1
→0,ρZ9←(+/Y9)÷ρ,Y9
Z9←'EMPTY'
```

```
QUILT
TRY:
:α←(¯2↑ 1 1 ,ρα)ρα
:E←α[⎕IO;]
:D←((1↑ρα),4)ρ' CTL DAT'
:→(D←0=PID ⎕SVOα,D,(ρα)ρE)/⎕14
:⍎E,'←⍳0'
:c← 1 0 1 0 ⎕SVC E
:→(0=c←ρ⍎E,'←ω')/0
:→((0,0)=1↑c←⍎E)/0
:→(1 2 ∧.≠1↑c)/⎕12
::(¯1↑c)↑'v'
:ω
:D←⎕SVRα
:'' CHKc
:(,' ',D≠'¨',α,'¨'),' SHARE OFFER FAILED'
CHK:
:→(0=ρ,α)/⎕3
:c←α=1↑ω
:→(∨/ω∈0,α)/0
:c←(⍳30)⍳1↑ω
:⍎ 21 ¯27[⎕IO+2=⎕NC 'OLE']↑'''TSIO ERROR ''
  ,(⍕ω),'' '',,OLE[⎕IO+c;]'
:⍎(0≤⌊/O)↑'→'
:∘
```

Compilation of *QUILT* by *FI* produces this form of *TRY*:

```
Z9←X9 TRY Y9;E;D
X9←(¯2↑ 1 1 ,ρX9)ρX9
E←X9[⎕IO;]
D←((1↑ρX9),4)ρ' CTL DAT'
→(D←0=PID ⎕SVO X9,D,(ρX9)ρE)/⎕14
⍎E,'←⍳0'
Z9← 1 0 1 0 ⎕SVC E
→(0=Z9←ρ⍎E,'← Y9 ')/0
→((0,0)=1↑Z9←⍎E)/0
→(1 2 ∧.≠1↑Z9)/⎕12
(¯1↑Z9)↑'v'
Y9
D←⎕SVR X9
'' CHK Z9
(,' ',D≠'¨',X9'¨'),' SHARE OFFER
  FAILED'
```

In a large system quilts are not a practical technique for localizing all functions that should not be global, and a simpler technique is needed. Here are two possible forms of extension of ∇:

```
∇'F X Y Z'
'W W' ∇ 'F F X Y Z'
```

The first example localizes functions *X*, *Y*, and *Z* in *F*, provided that (like *PCOPY*) the names are free in *F*. In the second example the repetition of *F* causes *X*, *Y*, and *Z* to override current objects, and *F* in turn is established in workspace *W* overriding any current *F* in *W*. Any combination of one and two workspace and global function names would be allowed.

□

Redefining Reduction Along an Empty Axis

Zeke Hoskin

Abstract

This note presents a consistent approach to evaluating the reduction of an array along an empty axis, giving results which agree with accepted values where these exist and also giving unique and consistent results for the Nand and Nor functions.

Note: For clarity, the body of this paper deals only with vectors; the extension to higher-rank arrays is straightforward.

- - - - -

Introduction

The reduction of an empty vector by a scalar dyadic function *F* is defined as the left- or right-identity value for *F*, or as an error if *F* has no identity [1,2]. This is theoretically unsatisfying and has the practical drawback that an application using reduction by Nand or Nor, which complete the set of ten nontrivial Boolean functions, must include code to deal with the empty case. This paper describes for each scalar dyadic function an identity which can be extended consistently to obtain a result for reduction of the empty vector.

Theory

For each APL scalar dyadic function *F*, we can find a "quasi-identity" element *Q* and a monadic function *G* such that, for any *X* within the range and the domain of *F*:

$$G\ Q\ F\ X \leftrightarrow X \qquad (1)$$

If *F* has a left identity, then *Q* is that identity element and *G* is the function that returns its argument unchanged. If *F* has a right-identity, then *G Q* will be equal to that identity element *I*. By the right-identity definition:

$$Q\ F\ I \leftrightarrow Q \qquad (2)$$

and by identity (1):

$$G\ Q\ F\ I \leftrightarrow I \qquad (3)$$

Then by (2) and (3):

$$G\ Q \leftrightarrow G\ Q\ F\ I \leftrightarrow I \qquad (4)$$

The scalar identity (1) can be extended into an identity on nonempty vectors, which can in turn be extended to the empty vector:

$$G\ F/Q,X \leftrightarrow F/X \qquad (5)$$
$$F/⍳0 \leftrightarrow G\ F/Q,⍳0 \leftrightarrow G\ Q \qquad (6)$$