APL 82, W. H. Janko, W. Stucky (ed.),
APL Quote Quad, Vol. 13 No. 1,

EXTENDING APL: WHAT MORE CAN A PROGRAMMER ASK FOR?

Dragan Bozinovic

IBM Canada Laboratory, Dept. 792
1150 Eglinton Avenue East
Don Mills, Ontario M3C 1H7
(416) 443-6711

1. Abstract

This paper explores certain underdeveloped parts of APL which ought to grow if APL is to qualify as implementation language for large and maintainable software systems. Following specific problems are discussed:

* Integrating APL with other parts of information processing environment.
  - Operating systems.
  - Programs written in other languages.
  - Data bases.
* Using independently developed functions and subsystems.
  - Problems with names.
  - Problems with space.
* Execution control of object attributes.
* Communications among functions.
  - Passing parameters.
  - Returning values.
  - Transferring control.
* Information hiding modules.
  - Packaging related functions.
  - Packaging data with functions.
  - Local functions.

2. Introduction

APL would benefit from unified treatment of different classes of objects, as pointed out by Crick in [1]. He proposes that workspaces, files, and defined functions, all be treated as arrays. (The word array as used in this paper includes non-simple or nested arrays.) He also proposes borrowing the access control mechanism (access matrices) from present APL file systems and permitting its application on any node of an array. This will allow N-way sharing of objects, now possible only with files. The proposal deserves more than an applause, it deserves work.

3. Approach

In order to reconcile the contradictory objectives of desirable uniformity and needed differentiation the following approach can be followed:

1. Define a common syntax for operations on all objects in the APL universe. Conceal the differences in internal representation from the programmers who are not interested in them.

2. Provide a lower level interface for programmers who need more control over the work of the system. This interface must also stay within the same common syntax. The interface will define a logical view of system's internal operations but in the form invariant to differences in implementation. This level will be oriented towards systems programmers.

3. Within the two previously defined interfaces, and within the same common syntax, provide functions for manipulation of objects outside of the APL universe -- objects normally controlled by the operating system. Again the interface with the operating system should have two levels corresponding with two levels of interface with APL:

1) The first level of interface with the operating system should be independent of the idiosyncrasies of any operating system. It will support only basic functions provided by the majority of operating systems. Programs written using this level will be portable across operating systems. APL will perform translation from the portable interface to the native interface of the host operating system.

2) The second level of interface with the operating system will be dependant on the operating system. It can be implemented as simply as accepting the commands of the operating system in their native form, passing them unchanged to the operating system, and returning the output of those commands in the form of APL objects.

Such integration into the outside environment should make APL suitable as an alternate command language for systems that allow user programs to replace the command interpreter, such as UNIX allows a user program to run as a "shell". It should also allow easy interface with programs written in other programming languages, and non-APL files and data bases, making APL suitable for data base query applications.

4. Outgrowing the Workspace

The concept of the workspace has greatly contributed to the popularity of APL as an interactive language, particularly by giving an easy way to restart interrupted sessions. This feature has to be maintained. However, the requirement that arguments of APL primitives must be resident in the active workspace is felt as a serious restriction imposed on the language. The library system has little chance to stand the test of time due to limitations like single level organization, and numbering used instead of naming.

5. Directories

I will use the word directory to denote the most general APL object. Outwardly, a directory is just an array indexed by name, containing other arrays, some of them being directories. The root directory contains the whole APL system, i.e. other system directories and directories of primary users. Directories of primary users may contain directories of secondary users, etc. Leaf directories contain no directories, only files, functions, and arrays of data. However, directories normally coexist with non-directory objects in the same directory. Potentially any directory can be used as a workspace, subject to restrictions by its access matrix. Loading of a workspace is replaced by choosing the global directory. This scheme is modeled after the directory structure of UNIX operating system [5]. It could be also thought of as a system of nested symbol tables.

Important attributes and roles of directories are described here:

PERMANENT	A directory that can be destroyed only explicitly.
TEMPORARY	A directory that is destroyed implicitly by a predefined system action.
IMPLICIT	A temporary directory created implicitly as a side effect of a system action.
INVOCATION	An implicit directory created by a function invocation and destroyed by the termination of that invocation.
SESSION	An implicit directory created by starting an interactive session and destroyed by its normal termination.
GLOBAL	Selected permanent directory in which global objects are created. Root of invocation directories.

5.1. Invocation Directories

When a function is invoked a new directory is created in the active directory. Names local to the function are defined in this new directory and marked as uninitialized. The new directory is then made the active directory. If a function is suspended due to error, the active directory is not changed. The directory created by function invocation is not a permanent directory. It lives as long as function invocation.

5.2. Resolving names

If a reference is made to a name not defined in the active directory, the parent directory is searched, then its parent directory, and so on until an object with that name is found or the search of the global directory is completed without finding the object with that name. The session directory is searched next.

If the name is not found in the directories searched by default, a program controlled list of directories is searched next. Distinguished variable

□DD Default Directories

is a vector of directory names. directories listed in □DD are searched in order. When the name is not found after session directory was searched, directories listed

in ⎕DD are searched in order. This allows access to system directories and directories of other users, and also allows adding and removing of directories with programs being tested.

If an object is found, it is used. If the search was unsuccessful and the name was used as the receiver of an assignment, an object with that name is normally created in the active directory unless the name was declared with scope attribute GLOBAL, when it is created in the global directory. Otherwise a VALUE ERROR is signalled.

If the receiver of an assignment is found in a directory for which the user does not have write permission, an object with the same name is created in the session directory. This allows testing of programs without updating critical directories.

5.3. Operations with Names

To distinguish operations performed on the name from those performed on the object of that name, APL primitive execute (⍎) can be extended as follows:

(⍎NAME)

* When used right of the assignment returns value of the object named by NAME.

* When used at the left of the assignment, returns the object named by the value of (character vector) NAME as a receiver.

```
      A←'B'
      B←3
      C←(⍎A)+1
      C
4
      (⍎A)←7
      B
7
```

The simple rule is: expressions evaluated on the right side of the assignment return values, evaluated on the left side of the assignment return receivers.

5.4. Qualifying Names

In addition to the default directory search, the proposed hierarchy of directories allows qualification of names with names of directories, subject to restrictions by the access matrices of directories. I use symbol ∘ to denote qualification, as in:

```
dir∘name
dir1∘dir2∘name
```

The leftmost directory is found by normal directory search unless the full qualification starting with the root directory is given. Full qualification is recognized by ∘ in front of the first directory name:

```
∘root∘dir1∘name
```

To increase the utility of name qualification we also need to know the names of several important directories that can be contained in distinguished variables:

```
⎕AD  Active  Directory
⎕GD  Global  Directory
⎕SD  Session Directory
```

A primitive or distinguished function should be provided to return the name of the parent directory when given a name of a directory as the argument. If the name passed as the argument is not fully qualified, it will be resolved using methods described above. Distinguished variables will contain names of directories fully qualified to the root directory.

Variables can be used as qualifiers instead of literal directory names. A variable used as a qualifier can be a vector of directory names. Directories will be searched in the order of appearance.

```
(⍎var_dir)∘name
```

6. Object Attributes

APL is a language with late binding. Value, shape, data type, and even object type of an object, can be changed at execution time. Two other important attributes, name scope and life span, are not accessible at execution time. Classes of objects are not defined completely, e.g. the only named constant in APL is label. Some attributes are restricted to particular object classes, e.g. protection exists only for functions and files but not for any other objects. Other attributes, like data type, may need to be expanded. Properties likely to be defined as object attributes, are: object class, name scope, life span, data type, protection, etc.

Most attributes of names and named objects that are kept in the symbol table entry for the object, are not accessible from APL programs. It would be useful to allow access to values of those attributes, as well as the assignment to them, through uniform syntax. If the distinguished function ⎕NC is allowed on the left of the assignment arrow we can change the name class by assignment, which is in APL the normal way to assign a value to anything that can have a value.

```
(⎕NC 'NAME') ← 2
```

Similar functions can be provided for name scope and life span, ⎕NS and ⎕LS, eliminating the necessity for declaring name scope on the function header line, and providing program control over these important attributes.

6.1. Object Class

Although all objects have the appearance of arrays, it is useful to be able to tell them apart when necessary. For example, as Crick has proposed in [1], functions can be kept in an executable tokenized form as nested arrays. He proposes execution of such arrays by an operator ∇. Having attributes allows an alternative approach without excluding the first.

```
(⎕NC 'NAME') ← 3
⍝ NAME is a function.
(⎕NC 'NAME') ← 2
⍝ NAME is a data array
⍝ and can be manipulated.
(⎕NC 'NAME') ← 3
⍝ NAME is a function again.
```

6.2. Name Scope

Name scope is fixed at function definition time. APL also provides an incomplete set of name scopes -- only two. Reference [6] identifies a complete set of five name scopes by considering following criteria:

1. Does the name defined and used in the active function refer to the same object as when used in the calling function. Yes -- global, no local.

2. Is the object referred to by this name in the active function accessible to the called function. Yes -- unmasked, no -- masked.

3. If the object referred to by this name is local to the active function, is the object, if such exists, that was known by the same name to the calling function accessible to functions called from the active function. Yes -- transparent.

The five name scopes will here be called:

1. GLOBAL
2. GLOBAL MASKED
3. LOCAL,
4. LOCAL MASKED, and
5. STRICTLY LOCAL or TRANSPARENT.

They could be assigned numbers and treated like object class:

```
(⎕NS 'name')←5
```

6.3. Life Span

The concept of life span is not clearly distinguished in APL. Local objects are always destroyed when defining function is normally terminated. Global objects can be destroyed only explicitly.

Explicit run-time definition of two life spans,

1. TEMPORARY
2. PERMANENT

is proposed in this paper. It will be useful to allow all combinations of life span and name scope. Particularly useful combination, presently missing from APL, will be PERMANENT life span with any of the LOCAL name scopes. It results in objects that are owned by the function but retain data between successive invocations of the function.

```
(⎕NS 'name')←5
(⎕LS 'name')←2
```

7. Defining Functions

Limitations placed on the number of objects that can be passed to a function (two) and returned as the results of a function (one) are harmful for the structuring of APL programs. I assume that these limitation will be lifted with the introduction of nested arrays by using the strand notation that will allow passing unlimited number of enclosed arguments and returning unlimited number of enclosed results. The syntax will be, hopefully, as simple as:

```
(A B C)←(L1 L2 L3) F R1 R2 R3

∇Z←L F R; L1 L2 L3 R1 R2 R3 Z1 Z2 Z3
(L1 L2 L3)←L
(R1 R2 R3)←R
 ...
Z←Z1 Z2 Z3
∇
```

Assuming that this problem is solved, I now turn to some other extensions to function definition that should make APL better suited for design of large, complex, or reliable software systems.

7.1. Label treated as a Vector

The concept of an array is a cornerstone of APL. However, label is now treated as a scalar integer constant. That is an anomaly in the language based on arrays. If a label appears on more than one line a reference to that label should give a vector of line numbers where that label appears. Label also should not be of data type INTEGER but rather of a separate data type LABEL and should identify the invocation in addition to the line number within the invoked function. This will help restrict manipulations that can be done with labels to those that are necessary and safe.

Branch will be legal only to labeled lines, not to any integer value. This will also weaken APL's dependency on line numbers and organization of functions into lines instead of statements.

The APL editor will automatically supply indices in the displayed version of function lines with "multiply defined" labels. Indices will start at ⎕IO and be assigned in increments of one. The following examples assume integer origin zero.

```
     ∇ F1; ⎕IO←0
[1]           ...
 ...
[7]  LBL[0]: ...
 ...
[11] LBL[1]: ...
 ...
[14] LBL[2]: F2
 ...
[17] LBL[3]: ...
 ...
     ∇
```

There will be a distinguished variable ⎕LINEL (LINE Label). It gives the value of the label vector for the label on the line/statement being executed. This value can be captured in a function invoked from that line by assigning ⎕LINEL to a local variable on the header line. This will allow easy definition of functions to simulate constructs of structured programming. That may be desirable for instruction in programming that will combine APL interactive debugging facilities with the style required in other programming languages, for prototyping applications to be translated into languages having such constructs, or simply for programming convenience.

```
    ∇F2;⎕IO←0;L←⎕LINEL
[1]  'SYNTAX ERROR
    ∇
```

When F1 is invoked and F2 is suspended due to SYNTAX ERROR, values of variables will be as defined by the following example:

```
      F1
SYNTAX ERROR
F2[1]: 'SYNTAX ERROR
                   ^
      ⎕LINEL

      L
 7 11 14 17
      ⎕LC
 1 14
      L ⍳ 1↑1↓⎕LC
2
```

Branching to the first and the last line defined by the label vector can be accomplished using one of the following origin independent branches:

First	Last
→label	
→1↑label	→¯1↑label
→⌊/label	→⌈/label

One index origin will have to be chosen for printing the listing of the function with multiply defined labels if they are to be annotated with indices. It will be the current value of ⎕IO in the workspace unless ⎕IO is localized and initialized in the header line of the function containing multiply defined labels, when this initial value will be used. Initialization expression must be restricted to a constant.

The ability to change ⎕IO within a function is more harmful than useful. Functions should be written entirely in one index origin that will be specified in the header line. This is however not a prerequisite for implementing label vectors as long as origin independent branching is used. Objects initialized on the header line should be assumed to be constants having name scope STRICTLY LOCAL. An attempt to change such an object within the function should result in CONSTANT ERROR.

With label defined as above, following branching patterns could be used:

```
loop:→(c)/1+¯1↑⎕LINEL
 ...
 ...
loop:→⎕LINEL
```

```
case:→(c1,c2,...,cn)/1+1↓¯1↓⎕LINEL
 ... no condition true
 ...
case:→¯1↑⎕LINEL
 ...
 ...
case:→¯1↑⎕LINEL
 ...
 ...
case:→¯1↑⎕LINEL
 ...
 ...
case:
```

7.2. Local Functions

Within one physical unit (for purposes of editing, copying, expunging, and other maintenance) several functions can be defined. Blocks can be defined within a function. Blocks have properties of functions regarding name scope and life span of objects. They are analogous to BEGIN-END blocks in ALGOL and PL/I. In this text word 'block' will be used to refer to block or function unless otherwise indicated. A block is defined as an in-line, parameterless, local function.

Labels are by default TEMPORARY constants STRICTLY LOCAL to the block in which they are defined. This prevents branching to labels in both enclosed and enclosing block. Alternatively, it may be useful to explicitly define labels as LOCAL, in the present APL sense, to the block in which they are defined. This still prevents branching to statements within an enclosed block. Exit from an inner block to a label in an outer block is now possible unless that label is redefined in the inner block.

Labels can also be passed as an argument to a function. Prerequisite for this is that label is not treated as an integer but as a distinguished data type LABEL, which carries information not only about the line number, but also about the function invocation to which that line number belongs. A label parameter can be used to make a para-normal exit to a point different from the point of invocation which can also be several levels up in invocation hierarchy. Granted, this will not simplify the management of the invocation stack, but it could be done since label is not treated as an integer any more. Label now points to the specific line in specific invocation. All invocations, subordinate to the invocation that receives control via branch to a label parameter, are purged before the transfer of control is performed. This, by the way, solves the problem of defining a function GOTO that simulates the branch arrow.

Although this goes against the simplified definition of structured programming, there are real programming situations where program clarity benefits from immediate return upon discovery of a para-normal condition. Tausworthe in [8] discusses handling of para-normal conditions within the framework of structured programming.

By nesting block definitions, local functions can be defined without the need to 'fix' them at execution time. This will eliminate present penalties for modular programming: large number of global functions or difficulty of debugging and editing dynamically created local functions.

```
∇ Z ← L F1 R; ⎕IO←0
  ...
  ∇ ; ⎕IO←0
  a Block.
    ...
    ...
  ∇
  ...
  ∇ Z ← L F2 R; ⎕IO←0
  a Local function.
    ...
    ...
  ∇
∇
```

7.3. Information Hiding

There can be any number of globally defined functions in a source function definition unit. There can be lines within the source function definition unit that are outside of the physical scope of any function. Those lines can be used to declare objects that are local to the source module but common to all top level (global) functions defined by the source module. Objects defined within the source unit but outside of the physical scope of any defined function, are within logical scope of all global functions defined by that source unit. If those objects are defined with life span PERMANENT and name scope STRICTLY LOCAL, they are owned by global functions defined in that source unit, i.e. all other functions can not change values of those objects and can access their values only as parameters. Such a cluster of defined functions is an elegant and safe implementation of information hiding module. Information to be hidden from the outside world but known to all functions in the cluster is defined in the same source module with functions but outside of the physical scope of any function.

Information hiding modules will have the structure outlined in the following example:

```
* * * TOP OF FILE * * *
[0] (⎕NS 'name_1') ← 5
    (⎕LS 'name_1') ← 2
    ... Declare attributes
    ... of owned objects,
    ... Name Scope as
    ... strictly local,
    ... Life Span as
    ... permanent.
    (⎕NS 'name_m') ← 5
    (⎕LS 'name_m') ← 2

[0] ∇Z←L function_1 R
     ...
     ...
[j] ∇

    ...
    ...

[0] ∇Z←L function_n R
     ...
     ...
[k] ∇
* * * END OF FILE * * *
```

8. Coroutines

The concept of routines cooperating at the same level of invocation hierarchy has been found useful and has been implemented in some high level languages. Reference [2] shows how coroutines can be implemented by using global variables without any changes

to APL syntax. An elegant way to implement coroutines by extending APL is proposed in this paper. It depends on the new system variable ⎕LASTL and the new operator which will here be called coinvocation operator and be represented by the right arrow (→) when it is not in the first position on the line.

Coroutines have one restriction -- they can not return values. In order to get any output from a coroutine it should be passed at least one argument by name. Alternatively, coroutines can be defined in the same information hiding module and communicate through commonly owned objects.

8.1. Coinvocation Operator

The coinvocation operator, when applied to a function, creates coinvocation of that function on the same nesting level as the invoking function. If the coinvocation of the function to which the coinvocation operator is applied already exists on the current nesting level, the control is transferred to the existing coinvocation without creating a new one. In order to accommodate the concept of coinvocations within an invocation, system variable ⎕LC will have to become a nested array.

In the list of coinvocations each coroutine is present as many times as it was resumed and in the order of resuming. This is important for tracing program execution. Following illustration represents the sequence of invocations and coinvocations top to bottom, left to right:

```
F1[A1]
F2[A2]
F3[A3]  F4[A4]  F5[A5]  F3[B3]
 .       .       .      F6[A6]
 .       .       .      F7[A7]
```

For the above example, ⎕LC can be represented as:

```
A7 A6 (B3 A5 A4 A3) A2 A1
```

where simple scalars represent return points for functions on the stack.

8.2. Last Line Executed

⎕LASTL is updated by the interpreter after each line is executed. It is an integer vector of shape zero or one showing the number of the last executed line. ⎕LASTL is PERMANENT but STRICTLY LOCAL to the function. Such combination allows keeping a distinct copy of this variable for each coinvocation and the retention of the value of that variable while the coinvocation is waiting to be resumed.

* When the function is invoked without coroutine operator the value of ⎕LASTL is an empty vector while the first line of the function is being executed.

* When the function is invoked with coroutine operator

  - If there is an activated coinvocation of this function within the most recent invocation, then the value of ⎕LASTL is the number of the line from which this function has last time transferred control to another coinvocation. The existent coinvocation is resumed.

  - Otherwise the value of ⎕LASTL is empty vector and coinvocation of this function is created within the most recent invocation.

Model for coroutine definition:

```
* * * TOP OF FILE * * *
(⎕NS 'PRODUCT')← 5 ⍝ Strictly local.
(⎕LS 'PRODUCT')← 2  ⍝ Permanent.

∇ L CONSUMER R; RESUME←⎕LASTL+1
  ...
 →RESUME
  ...
  A PRODUCER→ B
  CONSUME PRODUCT
  ...
  A PRODUCER→ B
  CONSUME PRODUCT
  ...
  A PRODUCER→ B
  CONSUME PRODUCT
  ...
∇

∇ L PRODUCER R; RESUME←⎕LASTL+1
  ...
 →RESUME
  ...
  PRODUCT ← PRODUCE
  A CONSUMER→ B
  ...
  PRODUCT ← PRODUCE
  A CONSUMER→ B
  ...
  A CONSUMER→ B
  PRODUCT ← PRODUCE
  ...
∇
* * * END OF FILE * * *
```

9. Conclusion

APL needs to incorporate some new concepts to make it more useful language for implementing large and maintainable applications. Some of the concepts that are becoming widely recognized as useful for disciplined approach to design and construction of software systems are proposed in this paper for inclusion in APL.

Acknowledgements

Criticism and advice from Jim Brown have helped me improve an early draft of this paper. Normand Montour did not allow me to forget about APL when it was contending for a place on my priority list.

References

1. Crick, Michael F. C., Should APL be a Declining Language, APL'81 Conference Proceedings, San Francisco, ACM Publication, pp. 83-88.

2. Giloi, W. K. and Hoffman, R., Adding a modern control structure to APL without changing the syntax, APL'76 Conference Proceedings (Ottawa 1976), ACM Publication, pp. 189-194.

3. Parnas, D. L., On the criteria to be used in decomposing systems into modules, CACM, Dec 1972.

4. Parnas, D. L., Designing software for ease of extension and contraction, IEEE Transactions on Software Engineering, March 1979.

5. Ritchie, D. M., and Thompson K., The UNIX Time-Sharing System, The Bell System Technical Journal, Volume 57, No. 6, July-August 1978, pp. 1905-1929. UNIX is a trademark of Bell Laboratories.

6. Seeds, G. M., Arpin, A., LaBarre M., Name scope control in APL defined functions, APL Quote Quad, Vol 8, No 4, June 1978, pp. 15-19.

7. Smith, Bob, Nested arrays, operators, and functions, APL'81 Conference Proceedings, San Francisco, ACM Publication, pp. 286-290.

8. Tausworthe, R. C., Standardized Development of Computer Software, Vol. 1, 1977, Prentice-Hall.