A TOOLBOX FOR APL PROGRAMMERS

Robert C. Metzger
APL Systems Consultant
Boston Office
I.P. Sharp Associates, Inc.
Suite 415, 148 State Street
Boston, MA USA 02109
617-523-2506

Abstract

A set of programming tools for APL programmers is described. One group of tools helps the reader understand APL by taking advantage of special features of communication terminals, as well as using nontraditional approaches toward program listings, and nonverbal communication methods. Another group of tools rewrites portions of APL programs in order to eliminate stumbling blocks to effective communication.

APL Programs as a Communication Medium

APL was originally developed as a notation for communicating algorithms.[1] Its effectiveness as a notation was demonstrated before a computer implementation was made available.[2] Unfortunately, many people who use the various implementations of APL do not write their APL programs so as to clearly communicate algorithms to other people, but merely to cause a digital computer to take some action.

The long-range solution to this problem is to educate APL users in the art of "courteous programming", ie. writing programs so as to help the people who will read them. The APL neophyte puzzling over a "one-liner" while reading a textbook, the APL user stumbling over someone else's "spaghetti logic" while trying to get a job done, and the professional APL programmer attempting to meet a deadline while doing battle with a large program containing many complicated statements and no comments are in need of more immediate help. This paper describes an integrated approach toward a set of APL programming tools written in APL, which are meant to make the meaning of APL programs more accessible to the reader.

A program written for one of the APL implementations communicates with two distinct recipients-- the language processor program and the various people who will read it in order to correct, modify, or replace it. The theoretical models of the communication process identify several possible areas where APL programming tools can aid the program reader in understanding the program at hand.

The SMCR model (Source-Message-Channel-Receiver) was conceived of by David Berlo. [3] The Source (program writer) is affected by four factors-- communication skill, knowledge, social system, and cultural system. In the APL programming environment, these could translate to APL programming technique, knowledge of the application, programming standards (adhered to or ignored) and the natural language used by the programmer. Similar factors affect the Receiver (program reader), except for the first, which is instead the ability to read APL programs. The Channel is a printed page or video screen where APL characters are displayed. The Message aspect of this model is of greatest importance to this paper.

The Message is "an ordered selection from an agreed set of signs intended to communicate information."[4] The set of signs are those represented on the APL keyboard, and such overstrikes as may be recognized by the APL system. The ordered selection will be based upon the syntactic rules of the APL language. The information to be communicated is a description of a process by which a person or a digital computer can manipulate a set of symbols according to the definitions of the APL functions so as to achieve a desired result.

A Message is influenced by five factors: elements, structure, encoding, content, and treatment. At the highest level, the elements of an APL program are the header, and program lines. The structure at this level is defined by placing the header at

the beginning, followed by program lines which are identified by integers starting at one and ascending in steps of one. At the next level, the elements of a program line are labels, statements, statement-separator symbols, and comments. The structure is defined by the rule that the label must be on the left, followed by zero or more APL statements, which may be blocked using the statement separator, followed by comment text which is separated from the rest of the line by the lamp symbol. At the next-lower level, the elements of an APL statement are user-defined names, system-defined names, primitive functions, operators, and punctuation symbols. The structure is defined by rules concerning the juxtaposition of operators, niladic, monadic, and dyadic functions, variables and punctuation symbols. At the lowest level, each of the above-mentioned elements are composed of different subsets of the APL character set, and each is constructed with its own set of rules. User-defined names, for example, are formed from alphanumeric characters, and must have an alphabetic character in the leftmost position. A complete set of APL programming tools must enable the programmer to focus on the elements and structure at any level desired.

The content of an APL program is the algorithm being expressed. Just as verbal messages expressed through the more traditional media need an overall theme, so too the basic purpose of an APL program should be expressible in a single sentence. Encoding and treatment refer to the particular ways a message is constructed so as to express it clearly, accurately, and concisely, and to minimize the possibility of misinterpretation. The practitioner of courteous programming consciously seeks an encoding of an algorithm which will make it easy for other people to understand. Certainly one reason that a set of APL programming tools is needed is that many APL programmers do not seek such encodings in their APL programs. Even the courteous programmers, however can benefit from the use of automated programming tools which aid the debugging and documentation process.

Table I

Emphatic/ Clarifying Principle	Newspaper Method	Public-Speech Method
1. Change Quantity of Signal	Headline in Bold Type	Raise or Lower Voice
2. Change Quality of Signal	Use Italic Type Font	Change Tone of Voice
3. Place Null Signal before Signal to be Emphasized	Surround with Blank Space	Pause
4. Use Position of Signal on Medium	Place Text in Center of Page	Not Applicable
5. Omit Less Important Parts Of Message		
6. Present Data both Verbally and Nonverbally	Use Tables, Graphs	Not Applicable
7. Use Non-Verbal Messages	Use Photos or Sketches	Use Gestures

Improving Communication Effectiveness

Since an APL program is partly a medium for communication between people, it is logical to investigate other media in search of methods for increasing communication effectiveness in APL programs. People learn primarily from visual and aural media, and so this search will focus on the newspaper and the public speech as representatives of these types of media.

Table I lists several methods by which a newspaper editor or a public speaker can emphasize, clarify, or reinforce a message.

Table II lists methods for emphasizing, clarifying, or reinforcing the elements, structure, coding, treatment, or content of an APL program, which are related to the principles listed in Table I.

Table II

Emphatic/ Clarifying Method	APL Terminal Providing This Feature
1a. Use Character Sets of Varying Sizes	Tektronix 4015, Anderson Jacobson 860
1b. Use Blinking Characters	Hewlett Packard 2641, HDS Concept APL
2a. Use Multi-Color Ribbons	Trendata 4000A, Diablo 1620 and others
2b. Use Inverse Video Display	Hewlett Packard 2641, HDS Concept APL
2c. Underline Text	All
3. Insert syntactically nonsignificant blanks between APL syntactic elements, and blank lines between program lines.	
4. Align line numbers, line labels, APL statements, and comments into columns.	
5. List only the statements in a program which have specified characteristics.	
6. Display an identifier cross-reference listing.	
7. Draw program flowcharts.	

Table III lists some principles by which the effectiveness with which a verbal message is communicated can be improved, and the applications of these principles to improving the readability of APL programs.

Psychological Set & Alternative Perspectives

The investigation of the newspaper and the public speech in search of ways to improve communication effectiveness and to remove stumbling blocks to effective communication has provided a number of suggestions for APL programming tools. There are various reasons that each individual tool should be effective. Is there a general reason why these tools as a set may prove effective?

Gerald Weinberg asserts that the "psychological set" of the program reader is the major impediment to distinguishing between what the mind believes a program says and what the program actually says.[5] The reader's psychological set may come from reading comments describing the program, or listening to someone else's ideas on how it works or should work, or having knowledge of the application context of the program, or other similar factors. Psychological set worsens the miserable job of reading computer programs written by discourteous programmers, because the mind sees what it expects, hopes for, or dreads, rather than what is actually there.

Weinberg suggests a number of ways in which programmers can be given alternative perspectives on programs.[6] Once the suggestions which pertain only to traditional programming languages, and especially to Weinberg's favorite, PL/I, are pruned out, the ideas which remain relate to similar ideas which the previous investigation brought out.

1) Display special texts in boldface, lower case, or underlined.
2) Display the program without comments.
3) Rename all variables.
4) List symbols in alphabetical order, together with references.
5) List the scope of symbols.

Weinberg states that the point of these procedures is not to change the physical meaning of the program, but to change its "psychological meaning". The purpose of such tools is "to make the machine help people take advantage of the immense psychological resources they have in overcoming their immense psychological shortcomings."[7]

Table III

Improvement Principle	Application to APL Programs
1. Standardize or Refine Vocabulary	1a. Rename Labels; 1b. Rename Local Variables.
2. Remove Grammatical Problems	2a. Break up "one-liners"; 2b. Unblock Unrelated Statements.
3. Change Order of Presentation	3a. Alphabetize Local Variables in Header; 3b. Restructure Flow of Control.
4. Remove Slang, Colloquialisms, Archaisms	4a. Convert Mixed Output; 4b. Convert I-beam Functions; 4c. Convert Keyword Functions.

Program-Analysis Tools

The tools described in this paper fall into two categories. The first group all produce some sort of display at the terminal which is derived from an APL program which is being analyzed. The second group modify the definition of an APL program in some way in order to improve readability. The tools belonging to the analysis group are described in this section.

The LIST tool allows the user to display part or all of the program listing. The program can be displayed in part or in its entirety, with or without comments.

The SPREAD tool displays the program with a blank inserted after each syntactic element. Program lines are separated by blank lines, and lines are folded so as to minimize splitting syntactic units. Output from this tool looks like the following.

```
     ∇ MATRIX←DELIMITERS VECTOMAT VECTOR;⎕IO
      ;EXPAND;LENGTH;ENDS

[1]  ⍝ RESTRUCTURES A VECTOR INTO MATRIX C
      ONTAINING SUBSTRINGS AS ROW

[2]  ⍝ RIGHT ARGUMENT- VECTOR CONTAINING S
      UBSTRINGS DELIMITED BY ANY ELEMENT
      OF THE LEFT ARGUMENT

[3]  ⍝ LEFT ARGUMENT-SCALAR OR VECTOR OF P
      OSSIBLE DELIMITING ELEMENTS

[4]  ⍝ RESULT- MATRIX WITH SUBSTRINGS AS R
      OWS, LEFT-JUSTIFIED

[5]  ⎕IO ← 1

[6]  ⍞ ( 2 ≠ ⎕NC 'DELIMITERS' ) / 'DELIM
      ITERS←1↑0⍴VECTOR'

[7]  VECTOR ← VECTOR , ( ~ ( ¯1 ↑ VECTOR
      ) ∈ DELIMITERS ) ⍴ DELIMITERS

[8]  ENDS ← ( VECTOR ∈ DELIMITERS ) / ⍳
      ⍴ VECTOR

[9]  LENGTH ← ¯1 + ENDS - 0 , ¯1 ↓ ENDS
```

```
[10]  EXPAND ← ( LENGTH ∘.≥ ⍳ ⌈/ LENGTH )
      , 1

[11]  MATRIX ← 0 ¯1 + ( ⍴ EXPAND ) ⍴ (
      , EXPAND ) \ VECTOR

      ∇
```

The XREF tool produces a diagnostic identifier cross-reference table. Several special actions may be specified by the user.

1) Literal strings to the right of the execute function in a statement may be included in the program text analyzed.

2) References to system functions, system variables, quad and quotequad may be included in the table.

3) Line-number references to identifiers which do not receive any diagnostic flag may be suppressed.

The table produced by the XREF tool consists of three columns reserved for diagnostic symbols, the actual identifiers, and the line numbers of references, together with symbols designating special references to the identifier. Three possible symbols can be placed next to the identifier. A star (asterisk) indicates that the identifier is not localized. This will happen if the identifier is misspelled, or it ought to be localized, or it refers to a global function or variable. A question mark indicates that the identifier is localized, but the first reference in the program is not an assignment. This will happen if a variable has not been initialized, or if a local function is defined, or if the logic of the program is not top-down, thus causing the assignment to occur on a line following the first reference. An exclamation mark indicates a definite error-- either a label has been localized in the header, or defined twice, or assigned a value, or had its value referred to by indexing.

The special reference indicators immediately follow the line number listed where the reference was made. The colon indicates a label definition. The left arrow indicates a value assignment. The left bracket indicates an indexed value reference and the left bracket followed by a left arrow indicates an indexed assignment. The arrow pointing down indicates an argument passed, and the arrow pointing up indicates a result returned. The special reference symbols allow the user to quickly locate all original definitions and subsequent modifications of a variable. Given the function definitions below, the XREF tool would tool produce the tables which follow.

```
      ∇ INDEX←VECTOR LAST∆INDEX ARRAY
[1]   ⍝ FINDS THE LAST OCCURRENCE OF THE ELE
      MENTS OF AN ARRAY IN A VECTOR
[2]   ⍝ RIGHT ARGUMENT- ANY ARRAY
[3]   ⍝ LEFT ARGUMENT-  ANY VECTOR
[4]   ⍝ RESULT- NUMERIC VECTOR OF SAME RANK
      AND SHAPE AS RIGHT ARGUMENT.
[5]   ⍝          CONTAINS POSITION OF LAST OC
      CURRENCE OF EACH ELEMENT OF RIGHT ARGUMEN
      T
[6]   INDEX←(⎕IO+⍴VECTOR)-(~⎕IO)+(⌽VECTOR)⍳
      ARRAY
[7]   INDEX←,INDEX
[8]   INDEX[(INDEX=⎕IO-1)/⍳⍴INDEX]←1+⍴VECTO
      R
[9]   INDEX←(⍴ARRAY)⍴INDEX
      ∇

      ∇ INDEX←VECTOR LAST∆INDEX ARRAY
      ARRAY     0↓ 6 9
      INDEX     0↓ 6← 7← 7 8[← 8 8 9← 9
      VECTOR    0↓ 6 6 8

      ∇ LIST←PROMPT CHOOSE VALID
[1]   ⍝ ALLOWS USER AT TERMINAL TO CHOOSE 1,
      SOME OR 'ALL' ELEMENTS OF A VECTOR
[2]   ⍝ RIGHT ARGUMENT- NUMERIC VECTOR SPECI
      FYING NUMBERS WHICH CAN BE ENTERED
[3]   ⍝ LEFT ARGUMENT- CHARACTER VECTOR PROM
      PT
[4]   ⍝ RESULT- NUMBERS ENTERED BY USER
[5]   ⍝ SUBROUTINES-ASK,DISPLAY,IF
[6]   CYCLE:LIST←PROMPT ASK EITHER
[7]    →EXIT IF 0∈⍴LIST ⍝ QUIT?
[8]    →ALL IF∧/'ALL'=3↑LIST ⍝ ALL COMMAND?
[9]    →EXIT IF~'***INVALID ENTRY***' DISPLA
      Y~∧/LIST∈VALID
[10]   →CYCLE
[11]  ALL:LIST←VALID
[12]  EXIT:
      ∇

      ∇ LIST←PROMPT CHOOSE VALID
      ALL         8 11:
    * ASK         6
      CYCLE       6: 10
    * DISPLAY     9
    * EITHER      6
      EXIT        7 9 12:
    * IF          7 8 9
      LIST        0↑ 6← 7 8 9 11←
      PROMPT      0↓ 6
      VALID       0↓ 9 11
```

The ALIGN tool displays the program listing with line numbers, line labels, APL statements, and line-end comments aligned in separate columns. The programmer may specify the maximum width to be used for each column.

The DISPLAY tool is provided to utilize the various display features of certain APL terminals. A number of different special display features are supported. Terminals which do not provide special display features have texts highlighted by placing a blank line beneath the line to be highlighted, and placing a "high minus" sign beneath the characters to be highlighted.

In order to control the special display features of the various terminals, the SHARP APL Arbitrary Output facility is used. This facility bypasses the standard APL output routines, and transmits codes directly to the terminal. The DISPLAY tool does the work of these system routines.

What parts of a program can be emphasized with these features? The user can specify that certain classes of syntactic units be highlighted with whatever alternative display feature the terminal in use supports. Alternatively, the user can specify that entire program lines which have specified characteristics be highlighted with the special display feature.

Any combination of the following classes of syntactic units may be highlighted:

1) Comments,
2) Literal constants,
3) Numeric constants,
4) System functions and variables,
5) Scalar dyadic functions,
6) Mixed functions,
7) Derived functions,
8) User-defined identifiers.

Any one of the following types of statements may be highlighted:

1) Flow of Control-- branch statements, execute statements and labels;
2) Terminal-Input/Output Statements-- explicit-output(quad), bare-output (quotequad), evaluated-input(quad), literal-input(quotequad), and mixed-output (semicolon) statements;
3) Origin-Dependent Statements-- use of indexing, index generator and index locator, roll, deal, dyadic transpose, grade up, grade down, and the axis operator;
4) Comparison-Tolerance-Dependent Statements-- use of floor, ceiling, membership, index locator, less than, greater than, less than or equal, greater than or equal, equality, inequality;
5) Statements containing a specified text;
6) Statements containing a specified name, outside of comments and literal constants.

Unfortunately, output produced by using the various special display features can not be reproduced in this paper. Highlighting the literal and numeric constants of a program with underscoring produces output like the following.

```
     ∇ VECTOR←NAME ALTER VECTOR;ADD;DELETE;TY
    PE
[1]  ⍝ ALLOWS USER TO ADD NUMBERS TO, OR DEL
    ETE NUMBERS FROM VECTOR
[2]  ⍝ RIGHT ARGUMENT- DATA VECTOR TO BE ALT
    ERED
[3]  ⍝ LEFT ARGUMENT- NAME OF LIST (CHAR VEC
    TOR).  MAY BE OMITTED.
[4]  ⍝ RESULT - VECTOR ARGUMENT, AS ALTERED
    BY USER
[5]  ⍝ SUBROUTINES- ASK
[6]  →(2≠⎕NC 'NAME')/'NAME←'''''

[7]  TYPE←(CHARS,NUMBERS)[⎕IO+NUMERIC VECTO
    R]
[8]  'THE CURRENT ',NAME,' LIST IS: '

[9]  ⎕←VECTOR
[10] ADD←'ADDITIONS TO LIST:' ASK TYPE

[11] DELETE←'DELETIONS FROM LIST:' ASK TYPE

[12] VECTOR←ADD,(~VECTOR∈DELETE)/VECTOR
[13] VECTOR←VECTOR[⍋VECTOR]
     ∇
```

The MASK tool extracts from a program statements which have specified characteristics and displays them. Only the program header and the relevant statements are displayed. Any one of the six types of statements which can be highlighted with the DISPLAY tool can be extracted with the MASK tool. Masking the flow of control statements of a program produces output like the following.

```
     ∇ REPORT←FN∆SUMMARY OPTION

[15] NAME∆LOOP:→(ROWS<I←I+1)⍴END∆NAME∆LOOP

[20]  →(0<1↑⍴IMAGE)⍴UNLOCKED

[22]  →NAME∆LOOP
[23] UNLOCKED:

[27]  →NAME∆LOOP
[28] END∆NAME∆LOOP:→(0∈⍴OPTION)⍴FORMAT
[29]  →(~∨/'⍒⍋'∈OPTION)⍴CHOOSE

[31] CHOOSE:
[32] FORMAT:→((¯5+⌊⎕PW÷2)<⍴TITLE)⍴ONE∆COLUM
    N

[39]  →EXIT
[40] ONE∆COLUMN:
[41] EXIT:

     ∇
```

The FLOWCHART tool does not produce the kind of flowcharts which the traditional flowcharting routines have produced. The traditional method draws polygons around all statements in the program, with lines and arrows connecting the flow-of-control statements. The problem with these charts is that too much detail is presented, so the programmer is looking at many non-control statements with square boxes drawn around them, which gives no more information than looking at the original program. Weinberg suggests that "a flow diagram, properly constructed, can help to gather nonlocal points of flow into a two-dimensional form so that they may be seen on one page."[8]

The FLOWCHART tool only displays flow-of-control statements and labels. By limiting the statements listed to flow-of-control, most program flowcharts can fit on a single page, giving the overall picture that the programmer desires. All branch statements are displayed. Since the execute function is used by some programmers to implement IF-THEN constructs, any statement which begins with the execute function can also be displayed. All other executable statements are not displayed. Contiguous blocks of one or more non-control statements are marked by a single box containing the

line numbers which are represented. Blocks of comment lines are treated similarly, except that a lamp symbol is displayed to distinguish the lines from executable statements. If several statements are grouped on a line, decimal fractions will be added to the base-line number to distinguish the statements when necessary.

One final tool was not suggested by the earlier investigations, but has proven to be very helpful to programmers debugging functions. The CHECK tool searches for 12 common syntax errors which are relatively easy to detect. These errors are:

1) Unmatched Parenthesis,
2) Unmatched Bracket,
3) Misplaced Branch Arrow,
4) Invalid Assignment,
5) Invalid Label,
6) Undefined Character,
7) Invalid Rightmost Character,
8) Invalid Leftmost Character,
9) Misplaced Null,
10) Misplaced Slash,
11) Misplaced Slope,
12) Misplaced Period.

Lines containing a given error are listed after the relevant error message. Given the program listed below, the CHECK tool will produce the output which follows.

```
    ∇ A FUNCTION B
[1]   C←~/B
[2]   1.5:A←
[3]   1→4
[4]   □←(A+B))+C
[5]   D←⍵5
    ∇

    ∇ A FUNCTION B
UNMATCHED PARENTHESES
[4]   □←(A+B))+C
MISPLACED BRANCH ARROW →
[3]   1→4
INVALID LABEL :
[2]   1.5:A←
UNDEFINED CHARACTER ⍵
[5]   D←⍵5
INVALID RIGHT-MOST CHARACTER ←
[2]   1.5:A←
MISPLACED SLASH /
[1]   C←~/B
```

Program-Improvement Tools

The tools described in this section all change some part of the definition of an APL program. They are not meant to change the program results, but rather the way the algorithm is expressed, in hopes of making it easier for people to understand.

The HEADER tool is quite simple. It rearranges the names localized in the header, other than the argument(s), result and function name, so that they are in alphabetical order.

The LABELS tool replaces all existing line labels with a set of generated labels. The new labels are generated by creating a vector of evenly spaced, increasing integers, and then an alphabetic character is used as a prefix to create legitimate APL names. The user can supply the prefix letter and the interval between numbers. This tool is helpful if the labels in a program were not chosen for meaning, and if there is no logical way for knowing where a particular label might be in the program. When labels have been carelessly chosen, this tool can also reduce Symbol-Table entries through standardization.

The RENAME tool systematically replaces all of the local variable and label names in a function. It provides four different methods for creating the new names, and thus can be used for a variety of purposes. If the names in a function are too long, a rare occurrence, the program can abbreviate them. Abbreviation can be done by simple truncation, or by removing duplicated consonants and vowels which are not in the first position. If the names are too similar, perhaps because they are just numbers prefixed by a letter, the program can generate names which are composed of a single letter repeated three or four times. These names, though meaningless, have proved sufficiently distinct to prevent mistyping and misreading. Finally, if a program has meaningful names, but needs to have "unlikely" names to avoid conflicts with the global environment, the same names, with alternating characters underscored, can be substituted.

The BREAKUP tool breaks a single statement with embedded assignments into several statements containing only one assignment. On APL implementations like SHARP APL which provide a statement-blocking feature, the new statements are placed on the same line the original statement came from. The BREAKUP tool handles assignments nested inside brackets or parentheses according to the following rules:

1) If one assignment is more deeply nested than another, the more deeply nested assignment will be placed first.
2) If two assignments are at the same level of nesting, the right-most will be placed first.

These rules will not work for all APL implementations, due to lack of a consistent treatment of embedded assignment. The program correctly avoids breaking up embedded assignments to quad and quotequad since these "primitive shared variables" behave differently depending upon which side of the assignment arrow they are on. Given the long statement listed below, the BREAKUP tool will produce the statements which follow.

```
→(0=1↑⍴POM←POM[J←((AB∈ 1 2 5)∧~((10×DIR[N2;1
   ])+PH)∈NBNC)/⍳1↑⍴POM←POM[J;6+⍳12];])⍴B0

POM←POM[J;6+⍳12]
J←((AB∈ 1 2 5)∧~((10×DIR[N2;1])+PH)∈NBNC)/⍳
   1↑⍴POM
POM←POM[J;]
→(0=1↑⍴POM)⍴B0
```

The UNBLOCK tool rearranges the statements in a program so that only one statement occurs on each line. This is, of course, only meaningful on systems like SHARP APL which provide a statement-blocking feature.

Three tools are provided to convert statements which use obsolete language constructs to use instead the current preferred form. The MIXED tool converts statements which use mixed output to use the monadic format function instead. The IBEAMS tool converts occurrences of I-beam functions, most of which can be directly converted to system functions and variables. Those which cannot be easily easily converted are marked with a comment calling for manual conversion. The KEYWORDS tool is only relevant to systems like SHARP APL which provided keyword functions before the introduction of system functions. It converts references to keyword functions to refer instead to the corresponding system functions. Given the mixed-output statement below, the MIXED tool will produce the statement which follows.

```
'ACCT ';DIR[I;1];' TOTALS FOR DEPT ';DEPT

⎕←(▼'ACCT '),(▼DIR[I;1]),(▼' TOTALS FOR DEPT
      '),(▼DEPT)
```

Finally, mention must be made of one tool which has not been implemented. The FLOW tool would rewrite the branching within a program so that only structured programming flow-of-control constructs would be used. It remains to be seen whether this is even possible, but it would certainly be very useful. As a first step towards this goal, however, a substitute is provided. The current FLOW tool converts a number of the most commonly used branching constructs from referring to absolute line numbers, the Line Counter, or I-beams 26 and 27, to instead refer to line labels. This is a necessary first step to solving the more general problem, and turns out both to be useful in conjunction with other programming tools, and to be a definite help in its own right.

Implementation Considerations

The programming tools described in this paper are currently available to I.P. Sharp employees on the SHARP APL timesharing system. The value of the tools has been such that versions have also been developed to work on other APL implementations. For the purposes of developing these alternative versions, the various APL implementations were divided into four classes. All of the versions of all the programs assume the existence of the common features of IBM's APL.SV and VSAPL, specifically the scan operator, the format and execute functions, system variables, and system functions. The first class of APL implementations provide only these features. The second class of APL implementations provide, in addition to the basics, a system function which returns a character-vector representation of an APL function, besides the matrix canonical representation provided by IBM implementations. The third class of implementations also provide a system function to allow direct transmission of codes to communication terminals, as well as the ability to place more than one APL statement on a program line, and the ability to place comments to the right of all executable statements on a program line. The fourth class of APL systems provides, in addition to the features mentioned above, a dedicated APL file system, a means of storing programs in internal-representation format on data files, and the ability to call user-defined functions both monadically and dyadically. Currently only SHARP APL is the fourth class.

There are versions of all of the programming tools described for implementation classes 3 and 4. The ALIGN and DISPLAY analysis tools, and the UNBLOCK improvement tool can not be provided for classes 1 and 2. The versions for class 2 will run somewhat faster than those for class 1, since the canonical matrix representation has to be converted into something resembling a vector representation in order to avoid explicit looping.

Since the programming tools will be used frequently, ideally they should reside in the workspace where programs are being developed or changed. In SHARP APL, large fixed-size workspaces are provided. The amount of space which the tools would occupy together in such workspaces makes storing them in the workspace, when not in use, an undesirable method. An alternative would be to force the programmer to copy the tools from a public library workspace before each use. This inconvenience would make it less likely that the programmer would use the tools. Both of these alternatives can be avoided, however, by using the flexible SHARP APL file subsystem, which can store one or more APL programs in internal-representation format in a file component. Two functions, ANALYZE and IMPROVE, are provided, which are small cover functions which open a public library file, fetch the necessary working functions, execute them, and erase them when done. This design has the advantage that it is no more expensive than copying each time, yet the programmer only copies the cover functions once, and thereafter the tools are immediately available whenever needed, without using much space in the workspace.

In the SHARP-APL versions, the programmer does not directly access the working tools, but calls the cover functions, specifying the functions to be analyzed, the tools to be employed, and control information through the arguments. In versions prepared for other implementations, the programmer calls the individual tools, specifying a single APL function, and necessary control information through the arguments.

Though some of the tools and ideas described in this paper have been implemented in some form on various APL systems, one of the important advantages of the approach taken in this paper is that the tools described here can be used as a group on a wide variety of APL systems. The programs are portable within implementation classes, and the tools are conceptually portable between most APL implementations.

Summary

The programming tools described in this paper were designed on the premise that APL is fundementally different from other programming languages. One of the most important differences is that APL has never been merely a means of getting a digital computer to do something, but is just as importantly a means of facilitating communication between people. APL programs should be written courteously, always considering how the person who reads the program will understand it. The programming tools described here can help a programmer understand APL programs which were not written in this manner, and improve understanding of those which were.

These programming tools were designed so that a programmer could view an APL program from a variety of perspectives, in order to overcome the problem of psychological set. The SMCR communication model, and relationships to other communication media were explored so as to provide as complete a set of tools as possible. One of the most important aspects of this approach is to provide a common set of tools which can be used on a variety of APL implementations.

As APL users give more consideration to those who will read their programs, some of these tools may become obsolete. Others may prove to be of such common interest that they will be provided as auxiliary processors, or built into APL function editors. Regardless of future developments, it is the author's hope that they will save the users of APL time and frustration as they develop, debug, and improve the APL programs which enable them to complete the larger tasks to which they address themselves.

References

[1] A.D. Falkoff and K.E. Iverson, "The Design of APL," IBM Journal of Research and Development XVII (1973).

[2] A.D. Falkoff, K.E. Iverson, and E. Sussenguth, "A Formal Description of System/360," IBM Systems Journal, III (1964).

[3] David K. Berlo, The Process of Communication: An Introduction to Theory and Practice (Holt, Rinehart & Winston, Inc., 1960).

[4] Colin Cherry, On Human Communication, (Cambridge: MIT Press, 1966), p. 307.

[6] Gerald Weinberg, The Psychology of Computer Programming (New York:, Van Nostrand Reinhold, 1971), pp. 162-163.

[6] Weinberg, pp. 266-267.

[7] Weinberg, p. 267.

[8] Weinberg, p. 265.