REPRESENTATION AND MANIPULATION OF DATA STRUCTURES IN APL

Harry Katzan, Jr.

Pratt Institute

ABSTRACT

Methods for the representation of complex data structures in APL, a programming language based on rectangular arrays and a multiplicity of functions, are presented. Data structures considered are: strings and sets, stacks and queues, tables, linked lists, and LISP-like structures. The material provides insight into the nature of data structures and should aid in establishing future requirements for primal and base languages.

INTRODUCTION

Modern computing systems tend to be complex compared with the simple word-oriented machines of fifteen years ago. Today, we hear of privileged operations, supervisor/problem states, interrupts, sophisticated I/O, etc. In spite of this, the primal language for using the computer (i.e., assembler language) has remained essentially the same - except for a few bells and whistles. Problem solvers and programmers, however, wish to use the machine in another way: with a higher-level language such as ALGOL, FORTRAN, or PL/I. Thusfar, the compiler has been the bridge between the languages of the user and the language of the computer. Because compiler and compilation costs are high and problem solvers and machine designers seem to be going in opposite directions, several researchers - namely: Bashkow, Sasson, and Kronfeld[1], Melbourne and Pugmire[2], Sugimto[3], and Weber[4] - have proposed and developed systems that directly execute the statements of a higher-level language. Recent advances in microprogramming and writable control store (e.g., see Husson[5]) indicate that the architecture of a computer using a higher-level language as a primal language is indeed feasible from both performance and cost standpoints.

The recent popularity of APL* has resulted in at least one APL machine (see Thurber and Myrna[6]) and led several researchers to conjecture on the possibility of implementing APL (or a subset of it) as a primal language. APL is also in widespread use as a problem solving language and the number of APL enthusiasts (in all areas of computer science) is growing rapidly. For both applications, the key question is: "How do we represent complex data structures in APL, a language based on rectangular arrays and a multiplicity of appropriate functions?" The answers should help to establish future requirements for primal and base languages.

This topic is the subject of this paper. The material should provide new insights into the nature and storage of complex data structures. Obviously, most of the concepts are already known. Yet, there is much benefit in providing a unified treatment of this important area of computer technology. The concepts, and functions as well, are presented by order of increasing complexity. In other words, the first few functions are relatively simple whereas the latter ones, especially those on LISP, are fairly obscure, using recursive functions and related techniques.

THE APL SYSTEM

Statements and Functions

The APL terminal system[7] combines Iverson's language[8] and the concept of time sharing to form an effective system for interactive computing. Input to APL takes one of two principle forms: statements in the APL language and system commands. System commands are used to address the APL terminal system itself and provide miscellaneous services that are outside the scope of the language itself. Statements in the APL language fall into 3 categories:

1. Specification statements such as

```
A←2×3+4
```

2. Branch statements, such as

```
→LOOP+2
```

3. Function definitions, such as

```
∇R←X PLUS Y
 R←X+Y
 ∇
```

Moreover, the system operates in two modes: the execution mode and the definition mode. In the execution mode, statements are executed

*APL stands for A Programming Language based on the book by K.E. Iverson, *A Programming Language*, Wiley, 1962.

immediately. In the definition mode, statements are stored as part of a function definition. System commands and functions can only be entered in the execution mode.

Specification and branch statements permit expressions as arguments. Expressions can be composed of constants, variables, monadic and dyadic functions, and parentheses in the usual sense. A right-to-left order of execution has been adopted. Table 1 contains the primitive scalar functions contained in APL. They also apply to array arguments on an element-by-element basis. Thus

4 10 18 ↔ 1 2 3×4 5 6

etc.

Composite Functions

The extension of the scalar dyadic functions to arrays are termed *composite functions*. Three functions fall into this category: reduction, inner product, and outer product. Reduction is written:

⊕/*A*

where ⊕ is a dyadic function and *A* is an array. Thus if *V*←3 2 9 1 4, then

19↔ +/*V*

Reduction also applies to rank-*n* arrays along a single coordinate and effectively reduces the number of coordinates by one.

Inner product, which is related to the ordinary matrix product, is written

Af.*gB*

where *f* and *g* are dyadic functions and *A* and *B* are arrays. The matrix product of conformable matrices *A* and *B* is denoted by

A+.×*B*

The outer product resembles the familiar cartesian product and is written

A∘.*fB*

where *A* and *B* are arrays and *f* is again a dyadic function. If *A*←1 2 3 and *B*←6 8 10, then *A*∘.+*B* yields the matrix:

```
7   9  11
8  10  12
9  11  13
```

<table>
<tr><th colspan="2">Monadic form fB</th><th>f</th><th colspan="2">Dyadic form AfB</th></tr>
<tr><th>Definition or example</th><th>Name</th><th></th><th>Name</th><th>Definition or example</th></tr>
<tr><td>+B ↔ 0+B</td><td>Plus</td><td>+</td><td>Plus</td><td>2+3.2 ↔ 5.2</td></tr>
<tr><td>-B ↔ 0-B</td><td>Negative</td><td>-</td><td>Minus</td><td>2-3.2 ↔ ¯1.2</td></tr>
<tr><td>×B ↔ (B>0)-(B<0)</td><td>Signum</td><td>×</td><td>Times</td><td>2×3.2 ↔ 6.4</td></tr>
<tr><td>÷B ↔ 1÷B</td><td>Reciprocal</td><td>÷</td><td>Divide</td><td>2÷3.2 ↔ 0.625</td></tr>
<tr><td rowspan="2"><table><tr><th>B</th><th>⌈B</th><th>⌊B</th></tr><tr><td>3.14</td><td>4</td><td>3</td></tr><tr><td>¯3.14</td><td>¯3</td><td>¯4</td></tr></table></td><td>Ceiling</td><td>⌈</td><td>Maximum</td><td>3⌈7 ↔ 7</td></tr>
<tr><td>Floor</td><td>⌊</td><td>Minimum</td><td>3⌊7 ↔ 3</td></tr>
<tr><td>*B ↔ (2.71828..)*B</td><td>Exponential</td><td>*</td><td>Power</td><td>2*3 ↔ 8</td></tr>
<tr><td>⍟*N ↔ N ↔ *⍟N</td><td>Natural logarithm</td><td>⍟</td><td>Logarithm</td><td>A⍟B ↔ Log B base A
A⍟B ↔ (⍟B)÷⍟A</td></tr>
<tr><td>|¯3.14 ↔ 3.14</td><td>Magnitude</td><td>|</td><td>Residue</td><td><table><tr><th>Case</th><th>A|B</th></tr><tr><td>A≠0</td><td>B-(|A)×⌊B÷|A</td></tr><tr><td>A=0,B≥0</td><td>B</td></tr><tr><td>A=0,B<0</td><td>Domain error</td></tr></table></td></tr>
<tr><td>!0 ↔ 1
!B ↔ B×!B-1
or !B ↔ Gamma(B+1)</td><td>Factorial</td><td>!</td><td>Binomial coefficient</td><td>A!B ↔ (!B)÷(!A)×!B-A
2!5 ↔ 10 3!5 ↔ 10</td></tr>
<tr><td>?B ↔ Random choice from ⍳B</td><td>Roll</td><td>?</td><td>Deal</td><td>A Mixed Function (See Table 3.8)</td></tr>
<tr><td>○B ↔ B×3.14159...</td><td>Pi times</td><td>○</td><td>Circular</td><td>See Table at left</td></tr>
<tr><td>~1 ↔ 0 ~0 ↔1</td><td>Not</td><td>~</td><td></td><td></td></tr>
<tr><td colspan="2" rowspan="2"><table><tr><th>(-A)○B</th><th>A</th><th>A○B</th></tr><tr><td>(1-B*2)*.5</td><td>0</td><td>(1-B*2)*.5</td></tr><tr><td>Arcsin B</td><td>1</td><td>Sine B</td></tr><tr><td>Arccos B</td><td>2</td><td>Cosine B</td></tr><tr><td>Arctan B</td><td>3</td><td>Tangent B</td></tr><tr><td>(¯1+B*2)*.5</td><td>4</td><td>(1+B*2)*.5</td></tr><tr><td>Arcsinh B</td><td>5</td><td>Sinh B</td></tr><tr><td>Arccosh B</td><td>6</td><td>Cosh B</td></tr><tr><td>Arctanh B</td><td>7</td><td>Tanh B</td></tr></table>Table of Dyadic ○ Functions</td><td>∧
∨
⍲
⍱</td><td>And
Or
Nand
Nor</td><td><table><tr><th>A</th><th>B</th><th>A∧B</th><th>A∨B</th><th>A⍲B</th><th>A⍱B</th></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table></td></tr>
<tr><td><
≤
=
≥
>
≠</td><td>Less
Not greater
Equal
Not less
Greater
Not Equal</td><td>Relations
Result is 1 if the relation holds, 0 if it does not:
3≤7 ↔ 1
7≤3 ↔ 0</td></tr>
</table>

Table 1. Primitive Scalar Functions[*]

Mixed Functions

The mixed functions in APL are designed for use with arrays and provide a variety of useful operations, such as:

- Generating a vector of integers.
- Finding the index of an element in a vector.
- Determining the size or shape of an array.
- Raveling an array or scalar to form a vector.
- Catenating vectors and rank-*n* arrays.
- Selecting or dropping elements of an array.
- Sequencing elements of a vector.
- Compressing and expanding an array.
- Reversal, rotation, and transposition.
- Set functions.
- Base value and representation functions.
- Random number generation.

Mixed functions are summarized in Table 2.

Indexing

A subscript in APL is termed an *index* and may be a scalar or an array. If *V* ← ¯7 3 9 6 5 1 4 3, then *V*[3 6 1]↔ 9 1 ¯7. Similarly, if *A* ← 2 2⍴2 7 3 1, i.e.,

$$A = \begin{pmatrix} 2 & 7 \\ 3 & 1 \end{pmatrix}$$

then

```
V[A]↔ 3  4
      9 ¯7
```

Also, if *B* ← 2 4⍴ ¯7 3 9 6 5 1 4 3, i.e.,

$$B = \begin{pmatrix} \bar{~}7 & 3 & 9 & 6 \\ 5 & 1 & 4 & 3 \end{pmatrix}$$

then *B*[2;3] = 4, *B*[;2] = 3 1, and *B*[2;] = 5 1 4 3.

Miscellaneous Comments

Since this paper contains a number of APL programs, several comments are necessary. First, the user's input is indented six spaces and the computer types beginning in the left hand margin. Next, if the *last* operation in a statement is not a branch or specification, then the result is typed at the terminal. Thus,

```
      A←10
      A+3
13
```

Name	Sign[1]	Definition or example[2]
Size	ρA	ρP ↔ 4 ρE ↔ 3 4 ρ5 ↔ ⍳0
Reshape	VρA	Reshape A to dimension V 3 4ρ⍳12 ↔ E 12ρE ↔ ⍳12 0ρE ↔ ⍳0
Ravel	,A	,A ↔ (×/ρA)ρA ,E ↔ ⍳12 ρ,5 ↔ 1
Catenate	V,V	P,⍳2 ↔ 2 3 5 7 1 2 'T','HIS' ↔ 'THIS'
Index[3 4]	V[A] M[A;A] A[A;.. ..;A]	P[2] ↔ 3 P[4 3 2 1] ↔ 7 5 3 2 E[1 3;3 2 1] ↔ 3 2 1 11 10 9 E[1;] ↔ 1 2 3 4 'ABCDEFGHIJKL'[E] ↔ ABCD E[;1] ↔ 1 5 9 EFGH IJKL
Index generator[3]	⍳S	First S integers ⍳4 ↔ 1 2 3 4 ⍳0 ↔ an empty vector
Index of[3]	V⍳A	Least index of A in V, or 1+ρV P⍳3 ↔ 2 P⍳E ↔ 5 1 2 5 3 5 4 5 5 5 5 5 4 4⍳4 ↔ 1
Take	V↑A	Take (drop) |V[I] first elements on coordinate I. (Last if V[I]<0) 2 3↑X ↔ ABC EFG ¯2↑P ↔ 5 7
Drop	V↓A	
Grade up[5]	⍋A	The permutation which would order A (ascending or descending) ⍋3 5 3 2 ↔ 4 1 3 2
Grade down[5]	⍒A	⍒3 5 3 2 ↔ 2 1 3 4
Compress[5]	V/A	1 0 1 0/P ↔ 2 5 1 0 1 0/E ↔ 1 3 5 7 9 11 1 0 1/[1]E ↔ 1 2 3 4 ↔ 1 0 1⌿E 9 10 11 12
Expand[5]	V\A	1 0 1\⍳2 ↔ 1 0 2 1 0 1 1 1\X ↔ A BCD E FGH I JKL
Reverse[5]	⌽A	⌽X ↔ DCBA HGFE LKJI ⌽[1]X ↔ ⊖X ↔ IJKL EFGH ABCD ⌽P ↔ 7 5 3 2
Rotate[5]	A⌽A	3⌽P ↔ 7 2 3 5 ↔ ¯1⌽P 1 0 ¯1⌽X ↔ BCDA EFGH LIJK
Transpose	V⍉A	Coordinate I of A becomes coordinate V[I] of result 2 1⍉X ↔ AEI BFJ CGK DHL 1 1⍉E ↔ 1 6 11
	⍉A	Transpose last two coordinates ⍉E ↔ 2 1⍉E
Membership	A∊A	ρW∊Y ↔ ρW E∊P ↔ 0 1 1 0 1 0 1 0 0 0 0 0 P∊⍳4 ↔ 1 1 0 0
Decode	V⊥V	10⊥1 7 7 6 ↔ 1776 24 60 60⊥1 2 3 ↔ 3723
Encode	V⊤S	24 60 60⊤3723 ↔ 1 2 3 60 60⊤3723 ↔ 2 3
Deal[3]	S?S	W?Y ↔ Random deal of W elements from ⍳Y

Table 2. Primitive Mixed Functions*

1. Restrictions on argument ranks are indicated by: S for scalar, V for vector, M for matrix, A for Any. Except as the first argument of $S\iota A$ or $S[A]$, a scalar may be used instead of a vector. A one-element array may replace any scalar.

2. Arrays used in examples: $P \leftrightarrow 2\ 3\ 5\ 7$ $\quad E \leftrightarrow \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$ $\quad X \leftrightarrow \begin{matrix} ABCD \\ EFGH \\ IJKL \end{matrix}$

3. Function depends on index origin.

4. Elision of any index selects all along that coordinate.

5. The function is applied along the last coordinate; the symbols ⌿, ⍀, and ⊖ are equivalent to /, \, and ⌽, respectively, except that the function is applied along the first coordinate. If $[S]$ appears after any of the symbols, the relevant coordinate is determined by the scalar S.

Notes to Table 2.

* Tables 1 and 2 and the above notes are reproduced from:
Falkoff, A.D., and K.E. Iverson, *APL*\360 *User's Manual*, Yorktown Heights, N.Y., IBM Corporation, Watson Research Center, 1968 (Also available as IBM form #GH20-0683-1)

Also, the function header statement needs further explanation. Consider,

∇*R*←*X ABC Y*;*I*;*J*

The del (∇) puts the APL system into the execution mode. *R* specifies an explicit result; *ABC* is the name of the function; *X* and *Y* are dummy variables (arguments); and *I* and *J* are local variables. Lastly, the quad symbol (⎕) or the quote-quad symbol (⍞) indicates input or output - depending on how it is used. *A*←⎕ denotes input and ⎕←*A* denotes output.

STRINGS AND SETS

The most primitive type of data structure, other than a scalar numeric data item, is the string - taken in this case to be a sequence of characters. In APL, a character string is stored as a vector so that a list of strings is stored as a two-dimensional array or an extra long vector. A set is stored in a similar manner but is restricted to either character or numeric data.

Substring

The SUBSTR function in PL/I, for example, is easily constructed in APL. The function uses a string name, an offset, and a length as follows:

```
SUBSTR(NME, LOC, LEN)
```

and is simulated in APL as shown in Figure 1. The function returns the value of the substring.

```
       ∇SUBSTR[⎕]∇
     ∇ R←S SUBSTR A
[1]    R←S[¯1+A[1]+⍳A[2]]
     ∇

       C←'TEA FOR TWO'
       C SUBSTR 5 3
FOR
       C SUBSTR 1 2
TE
```

Fig. 1 Substring

Alphabetic Sort

An amazingly simple APL program can be constructed to sort strings that are stored as a two-dimensional array. Each row of the matrix represents a distinct string as depicted in Figure 2. The function uses the base value function to compute an index for each row and then uses the grade up function to compute the permutation of indices that would order the rows in ascending sequence.

```
       ∇SORT[⎕]∇
     ∇ R←SORT A;S
[1]    S←' ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
[2]    R←A[⍋(2+⍴S)⊥⍉S⍳A;]
     ∇

       DATA
TEA FOR TWO
ALL COWS EAT GRASS
IMPOSSIBLE
SIGPLAN NEWS
MAGIC SQUARE

       SORT DATA
ALL COWS EAT GRASS
IMPOSSIBLE
MAGIC SQUARE
SIGPLAN NEWS
TEA FOR TWO
```

Fig. 2 Alphabetic sort

Pattern Matching and Replacement

The ability to search a given string for a sequence of characters had its foundations in Markov algorithms and is an important feature of the SNOBOL language. In SNOBOL, pattern matching and replacement has the general form:

```
STR PAT = REPL
```

where STR is the string reference, PAT is the pattern, and REPL is the replacement string. In the above skeleton, any of the constructs, except the string reference, can be omitted as required by a particular application. Two APL functions are presented in Figure 3. The first, *FIND*, gives the index of the first occurrence of one string in another. The second, *REPLACE*, replaces one sequence of characters with another. In the latter case, a dummy function *WITH* is used to give the function the appearance of being a statement in a problem-oriented language, i.e.,

```
STR REPLACE A WITH B
```

```
      ∇FIND[⎕]∇
    ∇ P←C FIND D
[1]   P←(∧/[1](¯1+⍳⍴C)⌽(C←,C)∘.=D)⍳1
    ∇

      ∇WITH[⎕]∇
    ∇ R←A WITH B
[1]   U135V←B
[2]   R←A
    ∇

      ∇REPLACE[⎕]∇
    ∇ R←STR REPLACE A;I;J
[1]   →((⍴STR)≥I←1↑,(A←,A) FIND(STR←,STR))/L1
[2]   →0,⍴R←STR
[3]  L1:R←STR[⍳I-1],U135V,STR[J+⍳(⍴STR)-J←¯1+I+⍴A]
    ∇

      TXT←'ALL COWS EAT GRASS'

      TXT REPLACE 'EAT' WITH 'CHEW'
ALL COWS CHEW GRASS

      TXT REPLACE 'COWS' WITH ''
ALL  EAT GRASS
```

Fig. 3 Pattern matching and replacement

Set Union and Intersection

The *membership* function in APL, written

$A \in B$

returns the value 1 if A is an element of B. The result has the same structure as the left argument. Thus *'TEA FOR TWO'*∊' ' yields the vector 0 0 0 1 0 0 0 1 0 0 0. The membership function is used in the union and intersection functions given in Figure 4.

```
        ∇UNION[⎕]∇
     ∇ R←U UNION V
[1]    R←U,(~V∊U←,U)/V←,V
     ∇

        ∇INTERS[⎕]∇
     ∇ R←U INTERS V
[1]    R←(U∊V)/U
     ∇

        'ABDGH' INTERS 'BGL'
BG
        1 3 7 UNION 3 4 5
1  3  7  4  5
```

Fig. 4 Set union and intersection

Character Translation

One of the most frequent problems in terminal-oriented systems involves a character translation based on the type of terminal on the other end of the telephone line. Although the operation is trivial conceptuatlly, it is often cumbersome unless the computer has an appropriate instruction. Figure 5 lists an appropriate *TRANS* function that utilizes the indexing facilities in APL.

STACKS AND QUEUES

Storage is maintained dynamically in APL and this feature is particularly useful for implementing stacks and queues. In each case, the object is represented as a vector but in contradistinction to most implementations, a *list pointer* is not required. Stack and queue functions use the *take* and *drop* functions in APL which are useful for operating on a list without decomposing it.

```
        ∇TRANS[⎕]∇
      ∇ B←TRANS A;A1;A2
[1]     A1←'-α⊥∩⌊∊_∇∆⍳∘''⎕|⊤○*?⍴⌈~↓∪⍵⊃↑⊂∧¨¯<≤=≥>≠∨'
[2]     A2←' ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 '
[3]     B←A2[A1⍳A]
      ∇

        TRANS '~∊α-_○⍴-~⍵○'
TEA FOR TWO
        TRANS '⍳|*○⌈⌈⍳⊥⎕∊'
IMPOSSIBLE
```

Fig. 5 Character translation

Queues

A *queue* is a data structure in which additions are made at one end and deletions are made at the other. It is frequently referred to as a *FIFO list*. Figure 6 contains functions for *QUE* and *DEQUE*, respectively.

```
        ∇QUE[⎕]∇
      ∇ QUE A
[1]     Q←Q,A
      ∇

        ∇DEQUE[⎕]∇
      ∇ R←DEQUE
[1]     R←1↑Q
[2]     Q←1↓Q
      ∇

        Q←⍳0

        QUE 45
        QUE 2
        QUE ¯17.1
        DEQUE
45
        DEQUE
2
        QUE 119
        DEQUE
¯17.1
```

Fig. 6 Queue functions

Stacks

A stack is a structure in which entries are made at the same end using a *last-in-first-out* algorithm. Stack functions are given in Figure 7.

```
        ∇PUSH[⎕]∇
      ∇ PUSH A
[1]     STACK←A,STACK
      ∇

        ∇PULL[⎕]∇
      ∇ R←PULL
[1]     R←1↑STACK
[2]     STACK←1↓STACK
      ∇

        STACK←⍳0

        PUSH 45
        PUSH 2
        PUSH ¯17.1
        PULL
¯17.1
        PULL
2
        PUSH 119
        PULL
119
```

Fig. 7 Stack functions

TABLES

A table is a set of ordered pairs (k_i, v_i) with unique first components k_i. Here the k_i's are taken to be numeric values while the values can be numeric values or character strings. An entry v_i is said to be associated with the key k_i. Table lookup involves determining, for a key k^1, the table entry (k_i, v_i) where

$$k^1 = k_i$$

The process makes available the required value v_i

Numeric Values

A numeric table is stored as an $(n\times2)$ matrix where the first column represents the keys and the second column represents the values. Given a key K and a table T, it is easily determined if that key is found in the table; in fact it is expressed as

$K\epsilon T[;1]$

Replacement, deletion, addition, and fetch functions are given in Figure 8.

Variable Length Character Values

Table management using variable-length character values represents more of a problem but is easily solved in APL. The keys are stored as a vector *ID* of numeric values. Character values are stored as a continuous string *TEXT* of characters. A supplementary vector *START* is also used to denote the position of each entry in *TEXT* corresponding to an element of *ID* and a vector *LENGTH* that gives the length of each variable-length entry. Consider the entries:

Key	*Value*
37	*'TEA FOR TWO'*
3	*'ALL COWS EAT GRASS'*
50	*'IMPOSSIBLE'*
14	*'SIGPLAN NEWS'*
159	*'MAGIC SQUARE'*

If these values were entered sequentially, they would be stored as follows:

```
    ID
37  3  50  14  159
    START
0  11  29  39  51  63
    LENGTH
11  18  10  12  12
    TEXT
TEA FOR TWOALL COWS EAT GRASSIMPOSSIBLESIGPLAN NEWSMAGIC SQUARE
```

Functions to store, fetch, and delete entries are given in Figure 9.

LINKED LISTS

Linked lists commonly exist in two forms: unidirectional lists and bidirectional lists, represented as follows:

```
      ∇CHECK[⎕]∇
    ∇ L←T CHECK K
[1]   L←K∊T[;1]
    ∇

      ∇INDEX[⎕]∇
    ∇ I←T INDEX K
[1]   I←T[;1]⍳K
    ∇

      ∇REPLACE[⎕]∇
    ∇ REPLACE V
[1]   →(TABLE CHECK V[1])/L1
[2]   →0,⍴⎕←'KEY NOT IN TABLE'
[3]  L1:TABLE[TABLE INDEX V[1];2]←V[2]
    ∇

      ∇ADD[⎕]∇
    ∇ ADD V
[1]   →(~TABLE CHECK V[1])/L1
[2]   →0,⍴⎕←'DUPLICATE KEY'
[3]  L1:TABLE←TABLE,[1] V
    ∇

      ∇FETCH[⎕]∇
    ∇ R←FETCH K
[1]   →(TABLE CHECK K)/L1
[2]   →0,⍴⎕←'KEY NOT IN TABLE'
[3]  L1:R←TABLE[TABLE INDEX K;2]
    ∇

      ∇DELETE[⎕]∇
    ∇ DELETE K;I
[1]   →(TABLE CHECK K)/L1
[2]   →0,⍴⎕←'KEY NOT IN TABLE'
[3]  L1:TABLE←(((I-1),2)↑TABLE),[1]((-((1↑⍴TABLE)-I←TABLE INDEX K)),2)↑TABLE
    ∇
```

Fig. 8 Functions for use with tables with numeric values

```
        TABLE
    3     34
    2     67
    8     32
    9    112
    5     55

        REPLACE 8 75

        TABLE
    3     34
    2     67
    8     75
    9    112
    5     55

        ADD 13 0

        TABLE
    3     34
    2     67
    8     75
    9    112
    5     55
   13      0

        FETCH 2
67
        FETCH 45
KEY NOT IN TABLE

        DELETE 9

        TABLE
    3   34
    2   67
    8   75
    5   55
   13    0

        DELETE 41
KEY NOT IN TABLE
```

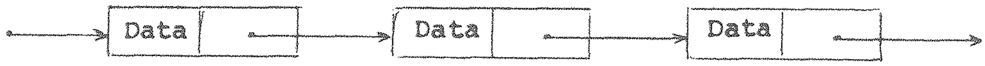
Fig. 8 (Continued)

```
      ∇INIT[⎕]∇
    ∇ INIT
[1]   ID←LENGTH←TEXT←⍳0
[2]   START←,0
    ∇

      ∇STORE[⎕]∇
    ∇ STORE;I;A
[1]   'ENTER INTEGER ID FOLLOWED BY TEXT ON THE NEXT LINE'
[2]   →(0=I←⎕)/0
[3]   →(0=ρA←⍞)/0
[4]   LENGTH←LENGTH,ρA
[5]   ID←ID,I
[6]   START←START,ρTEXT←TEXT,A
[7]   →2
    ∇

      ∇FETCH[⎕]∇
    ∇ FETCH LIST;IND;I;L
[1]   L←ρIND←ID⍳LIST←,LIST
[2]   →(1=∨/IND>ρID)/ERR
[3]   I←0
[4]  LOOP:→(L<I←I+1)/0
[5]   TEXT[START[IND[I]]+⍳LENGTH[IND[I]]]
[6]   ⎕←''
[7]   →LOOP
[8]  ERR:'INVALID ID'
    ∇

      ∇DELETE[⎕]∇
    ∇ DELETE KEY;I
[1]   →((ρID)≥I←ID⍳KEY)/GO
[2]   →0,ρ⎕←'INVALID ID'
[3]  GO:TEXT←TEXT[⍳START[I]],TEXT[J+⍳(ρTEXT)-J←START[I]+LENGTH[I]]
[4]   ID←ID[⍳I-1],ID[I+⍳(ρID)-I]
[5]   START←START[⍳I-1],((START[I+⍳(ρSTART)-I])-LENGTH[I])
[6]   LENGTH←LENGTH[⍳I-1],LENGTH[I+⍳(ρLENGTH)-I]
    ∇
```

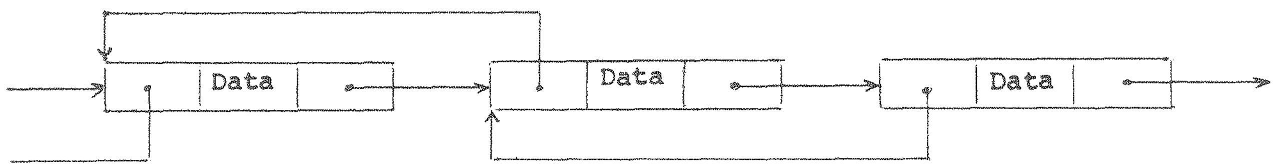
Fig. 9 Functions for use with tables with variable-length values

```
      INIT
      STORE
ENTER INTEGER ID FOLLOWED BY TEXT ON THE NEXT LINE
⎕:
      37
TEA FOR TWO
⎕:
      3
ALL COWS EAT GRASS
⎕:
      50
IMPOSSIBLE
⎕:
      14
SIGPLAN NEWS
⎕:
      159
MAGIC SQUARE
⎕:
      0

      FETCH 14 3
SIGPLAN NEWS

ALL COWS EAT GRASS

      ID
37  3  50  14  159

      DELETE 14
      FETCH 14
INVALID ID
      FETCH 50
IMPOSSIBLE
```

Fig. 9 (Continued)



Unidirectional List



Bidirectional List

In APL, the data part of a linked-list is stored as numeric or character data, as required. Pointer data is stored as a numeric array with indices to preceding and succeeding nodes, as required.

Unidirectional Lists

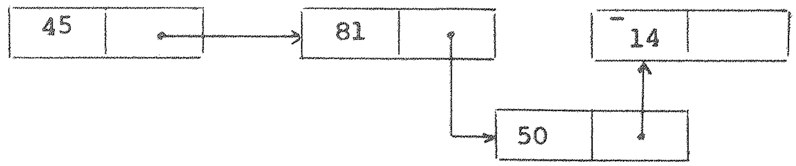
Consider the numeric list

It is represented in APL as:

LIST =

45	2
81	3
¯14	0

Adding a node after the second one is depicted as follows:



and in APL as:

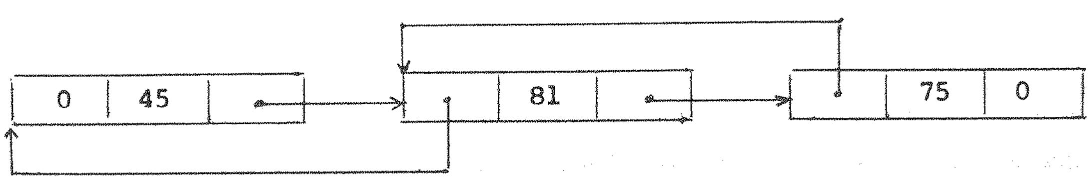
LIST =

45	2
81	4
¯14	0
50	3

Thus, deletions and additions are made without requiring that other data items be moved. Figure 10 gives functions for listing, adding, and deleting node elements. Although boundary conditions have not been satisfied in all cases to preserve clarity of exposition, the functions demonstrate the flexibility inherent in APL and the effective use of dynamic storage.

Bidirectional Lists

Bidirectional lists are similar to their unidirectional counterparts but contain backward pointers as shown previously. Backward pointers facilitate deletion and require only that the location of the node to be deleted be known. The list:



is represented in APL as:

```
      ∇LIST[⎕]∇
    ∇ R←LIST L;I;J
[1]   I←J←1
[2]  LOOP:→(L[J;2]=0)/PRINT
[3]   I←I,J←L[J;2]
[4]   →LOOP
[5]  PRINT:R←L[I;1]
    ∇

      ∇INSERT[⎕]∇
    ∇ INSERT N
[1]   ⍝ N[1] NODE AFTER WHICH NEW NODE SHOULD BE INSERTED
[2]   ⍝ N[2] NEW NODE
[3]   L[N[1];2]←1↑⍴L←L,[1] N[2],L[N[1];2]
    ∇

      ∇INDEXOF[⎕]∇
    ∇ R←INDEXOF A
[1]   R←L[;1]⍳A
    ∇

      ∇PRED[⎕]∇
    ∇ R←PRED I
[1]   R←L[;2]⍳I
    ∇

      ∇DELETE[⎕]∇
    ∇ DELETE N
[1]   ⍝ N[1] NODE TO BE DELETED
[2]   ⍝ N[2] PRECEDING NODE
[3]   L[N[2];2]←L[N[1];2]
    ∇

      ∇APPEND[⎕]∇
    ∇ APPEND A;I;J
[1]   I←1
[2]  L1:→((I←L[J←I;2])≠0)/L1
[3]   L←L,[1](A,0)
[4]   L[J;2]←1↑⍴L
    ∇
```

Fig. 10 Manipulation of unidirectional linked lists

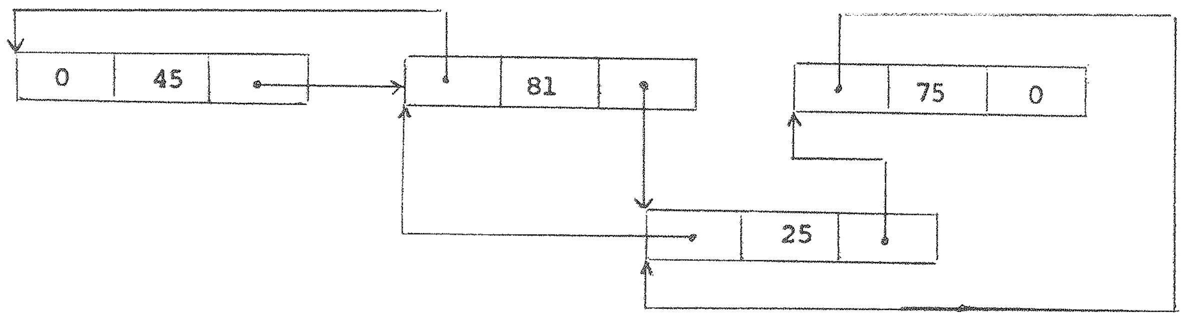
```
        L
  45    2
  81    3
 ¯14    0
        LIST L
45  81  ¯14
        INSERT 2 50
        L
  45    2
  81    4
 ¯14    0
  50    3
        LIST L
45  81  50  ¯14
        INDEXOF 81
2
        PRED 3
4
        PRED INDEXOF 50
2
        DELETE 2 1
        LIST L
45  50  ¯14
        APPEND 25
        LIST L
45  50  ¯14  25
```

Fig. 10 (Continued)

LIST =

0	45	2
1	81	3
2	75	0

Again, an additional node is depicted schematically as:



and in APL as:

LIST =

0	45	2
1	81	4
4	75	0
2	25	3

Sample APL functions for bidirectional linked lists are given as Figure 11.

LISP-LIKE STRUCTURES

The LISP language, developed by McCarthy[9] and discussed by Hopgood[10] and Katzan[11], presents data structures that are more complicated to represent and to effectively process.

```
    ∇LIST[⎕]∇
  ∇ R←LIST L;I;J
    I←J←1
   LOOP:→(L[J;3]=0)/PRINT
    I←I,J←L[J;3]
    →LOOP
   PRINT:R←L[I;2]
  ∇

    ∇APPEND[⎕]∇
  ∇ APPEND A;I;J
    I←1
   L1:→((I←L[J←I;3])≠0)/L1
    L←L,[1] J,A,0
    L[J;3]←1↑ρL
  ∇

    ∇INSERT[⎕]∇
  ∇ INSERT N;I
    ⍝ N[1] NODE AFTER WHICH NEW NODE SHOULD BE INSERTED
    ⍝ N[2] NEW NODE
    →(L[N[1];3]≠0)/L1
    APPEND N[2]
    →0
   L1:L[L[I;3];1]←L[N[1];3]←I←1↑ρL←L,[1] N,L[N[1];3]
  ∇

    ∇INDEXOF[⎕]∇
  ∇ R←INDEXOF A
    R←L[;2]⍳A
  ∇

    ∇PRED[⎕]∇
  ∇ R←PRED I
    R←L[;3]⍳I
  ∇

    ∇DELETE[⎕]∇
  ∇ DELETE I
    ⍝ I INDEX OF NODE TO BE DELETED
    L[L[I;1];3]←L[I;3]
    →(L[I;3]=0)/0
    L[L[I;3];1]←L[I;1]
  ∇
```

Fig. 11 Manipulation of bidirectional linked lists

```
       L
    0  45   2
    1  81   0
       APPEND 75
       L
    0  45   2
    1  81   3
    2  75   0
       INSERT 2 25
       L
    0  45   2
    1  81   4
    4  75   0
    2  25   3
       LIST L
45  81  25  75
       INDEXOF 75
3
       PRED 3
4
       APPEND 90
       L
    0  45   2
    1  81   4
    4  75   5
    2  25   3
    3  90   0
       LIST L
45  81  25  75  90
       DELETE INDEXOF 25
       LIST L
45  81  75  90
```

Fig. 11 (Continued)

Let the LISP register, depicted as

car	cdr

be represented in APL as a matrix of the form

type	index	car
type	index	cdr

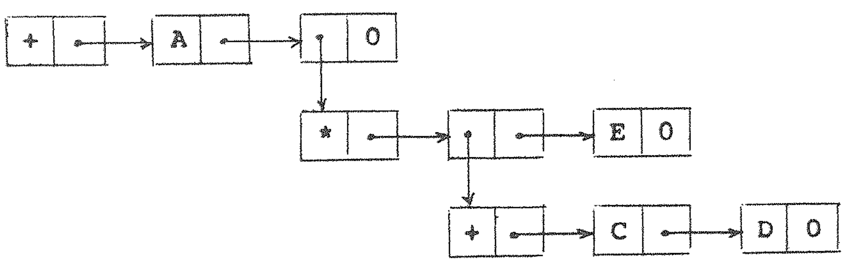
where

type	*structure*
0	composite symbol
1	atomic symbol
2	null symbol

Atomic symbols are stored as single characters in a character array. Composite symbols are stored as a rank-3 array - in this case named *LIST*. The character array is appropriately named *DATA*. Thus the LISP representation of

+A*+CDE

and depicted as



would be stored in the APL version as:

```
          DATA
      +A**+CDE

          LIST
       1  1
       0  2

       1  2
       0  3

       0  4
       2  0

       1  3
       0  5

       0  7
       0  6

       1  7
       2  0

       1  4
       0  8

       1  5
       0  9

       1  6
       2  0
```

APL functions (Figure 12) are developed to perform the following LISP-lime operations:

1. print
2. cons
3. car
4. cdr
5. atom
6. nil
7. enter data

CONCLUSIONS

Although the preceding discussion is not, and is not meant to be, a complete treatment of data structures and associated processing, it is perhaps indicative of the functions that programmers actually program and of structures that language designers consider. There is no intent here to debate whether the data structures provided in APL are sufficiently primitive to build more complicated structures or whether the APL primitive

```
      ∇ENTER[⎕]∇
    ∇ R←ENTER A
[1]   R←1,ρDATA←DATA,A
    ∇

      ∇CONS[⎕]∇
    ∇ R←A CONS B
[1]   R←1↑ρLIST←LIST,[1] A,[0.5] B
    ∇

      ∇CAR[⎕]∇
    ∇ R←CAR I
[1]   R←LIST[I;1;]
    ∇

      ∇CDR[⎕]∇
    ∇ R←CDR I
[1]   R←LIST[I;2;]
    ∇

      ∇NIL[⎕]∇
    ∇ R←NIL
[1]   R← 2 0
    ∇

      ∇ATOM[⎕]∇
    ∇ R←ATOM V
[1]   R←1=1↑V
    ∇

      ∇INIT[⎕]∇
    ∇ INIT
[1]   DATA←⍳0
[2]   LIST← 0 2 2 ρ0
    ∇
```

Fig. 12 Manipulation of LISP-like structures

```
      ∇PRINT[⎕]∇
    ∇ R←PRINT A
[1]    →(1=ρ,A)/L1
[2]    →(2=ρ,A)/L2
[3]    →0,ρR←'DOMAIN ERROR IN PRINT'
[4]   L1:→0,ρR←LPRINT A
[5]   L2:→(0=1↑A)/L3
[6]    →(1=1↑A)/L5
[7]    →ρR←ι0
[8]   L3:→0,ρR←LPRINT A[2]
[9]   L5:→0ρR←DATA[A[2]]
    ∇

      ∇LPRINT[⎕]∇
    ∇ R←LPRINT A
[1]    →((LIST[A;1;1]=1)∧LIST[A;2;1]=1)/L1
[2]    →((LIST[A;1;1]=0)∧LIST[A;2;1]=1)/L2
[3]    →((LIST[A;1;1]=1)∧LIST[A;2;1]=0)/L3
[4]    →((LIST[A;1;1]=0)∧LIST[A;2;1]=0)/L4
[5]    →((LIST[A;1;1]=1)∧LIST[A;2;1]=2)/L5
[6]    →((LIST[A;1;1]=0)∧LIST[A;2;1]=2)/L6
[7]    →ρR←ι0
[8]   L1:→0,ρR←DATA[LIST[A;1;2]],DATA[LIST[A;2;2]]
[9]   L2:→0,ρR←(LPRINT LIST[A;1;2]),DATA[LIST[A;2;2]]
[10]  L3:→0,ρR←DATA[LIST[A;1;2]],LPRINT LIST[A;2;2]
[11]  L4:→0,ρR←(LPRINT LIST[A;1;2]),LPRINT LIST[A;2;2]
[12]  L5:→0,ρR←DATA[LIST[A;1;2]]
[13]  L6:→0,ρR←LPRINT LIST[A;1;2]
    ∇
```

Fig. 12 (Continuation 1)

```
        PRINT 1
+A*+CDE

        CAR 1
1   1

        PRINT CAR 1
+

        PRINT CDR 1
A*+CDE

        PRINT CAR 3
*+CDE

        ATOM CAR 4
1

        PRINT CAR 4
*

        INIT

        I←(ENTER '+') CONS COMP (ENTER 'X') CONS COMP
           (ENTER 'Y') CONS NIL

        PRINT I
+XY

        LIST
   1   1
   2   0

   1   2
   0   1

   1   3
   0   2

        DATA
YX+
```

Fig. 12 (Continuation 2)

```
      LOC←(ENTER '*') CONS COMP (COMP I) CONS COMP
          (ENTER 'Z') CONS NIL

      PRINT LOC
*+XYZ

      LIST
  1  1
  2  0

  1  2
  0  1

  1  3
  0  2

  1  4
  2  0

  0  3
  0  4

  1  5
  0  5

      DATA
YX+Z*
```

Fig. 12 (Continuation 3)

functions are sufficiently rich. It cannot be ignored, however, that the functions presented are amazingly simple - that is, considering the amount of programming that ordinarily would be required in assembler language or most other languages. It seems that we as language designers should be as interested in the base language upon which sophisticated structures can be built as we are on the data structures themselves. In this way, our labors may affect, significantly, the computing machines of tomorrow.

REFERENCES

1. Bashkow, T.R., A. Sasson, and A. Kronfeld, "System design of a FORTRAN machine," *IEEE Transactions on Electronic Computers*, EG-16 (August, 1967), p. 485-499.
2. Melbourne, A.J., and J.M. Pugmire, "A small computer for the direct processing of FORTRAN statements," *Computer Journal*, 8(1968), p. 24-27.
3. Sugimto, M., "PL/I reducer and direct processor," *Proceedings of the 24th ACM National Conference*, (1969), p. 519-538.
4. Weber, H., "A microprogrammed implementation of EULER on IBM 360/30," *Communications of the ACM*, (September, 1967), p. 549-558.
5. Husson, S., *Microprogramming: Principles and Practices*, Englewood Cliffs, N.J., Prentice-Hall, Inc., 1970.
6. Thurber, K.J. and J.W. Myrna, "System design of a cellular APL computer," *IEEE Transactions on Computers*, Vol. c-19, Number 4 (April, 1970), p. 291-303.
7. Katzan, H., *APL Programming and Computer Techniques*, New York, Van Nostrand Reinhold Co., 1970.
8. Iverson, K.E., *A Programming Language*, New York, John Wiley and Sons, Inc., 1962.
9. McCarthy, J., *et al., LISP 1.5 Programmer's Manual*, Cambridge, Mass., The M.I.T. Press, 1962.
10. Hopgood, F.R.A., *Compiling Techniques*, New York, American Elsevier Publishing Company, Inc., 1969.
11. Katzan, H., *Advanced Programming: Programming and Operating Systems*, New York, Van Nostrand Reinhold Co., 1970.