JULY 1977

HEWLETT-PACKARD JOURNAL

Small Computer System Supports Large-Scale Multi-User APL

Powerful, interactive APL is now available for the multilingual HP 3000 Series II Computer System. A special terminal displays the APL character set.

by Kenneth A. Van Bree

APL (A PROGRAMMING LANGUAGE) is an interactive language that allows access to the full power of a large computer while maintaining a user interface as friendly as a desktop calculator. APL is based on a notation developed by Dr. Kenneth Iverson[1] of IBM Corporation over a decade ago, and has been growing in popularity in both the business and scientific community. The popularity of APL stems from its powerful primitive operations and data structures, coupled with its ease of programming and debugging.

Most versions of APL to date have been on large and therefore expensive computers. Because of the expense involved in owning a computer large enough to run APL, most of the use of APL outside of IBM has been through commercial timesharing companies. The introduction of APL\3000 marks the first time a large-machine APL has been available on a small computer. APL\3000 is a combination of software for the HP 3000 Series II Computer System[2] and a CRT terminal, the HP 2641A, that displays the special symbols used in APL. The terminal is described in the article beginning on page 25.

Although the HP 3000 is normally considered a small computer, APL\3000 is not a small version of the APL language (see page14). As a matter of fact, APL\3000 has many features that have never been available before, even on the large computers. For example, although APL\3000 looks to the user just like an interpreter, it is actually a *dynamic compiler*. Code is compiled for each statement as it is encountered; on subsequent executions of the statement, if the compiled code is valid, it is re-executed. By eliminating the interpretive overhead, a speedup on the order of a factor of ten can be obtained in some cases, although the speedup is dependent on the amount of computation involved in the statement.

The basic data type of APL is an array, which is an ordered collection of numbers or characters. *Subscript calculus*, as defined by Philip Abrams,[3] is a method of selecting portions of an array by manipulating the descriptors that tell how the array is stored. The use of subscript calculus in the dynamic compiler allows computation to be avoided in many cases, and eliminates the need for many temporary variables to store intermediate results.

One problem that has always plagued APL users is the limited size of most APL workspaces. A workspace in APL is a named data area that contains all the data variables and functions that relate to a particular

Cover: *In the foreground, Model 2641A APL Display Station demonstrates its role as the principal user interface for APL\3000, an enhanced version of APL (A Programming Language) that is now available on the HP 3000 Series II Computer System in the background.*

In this Issue:

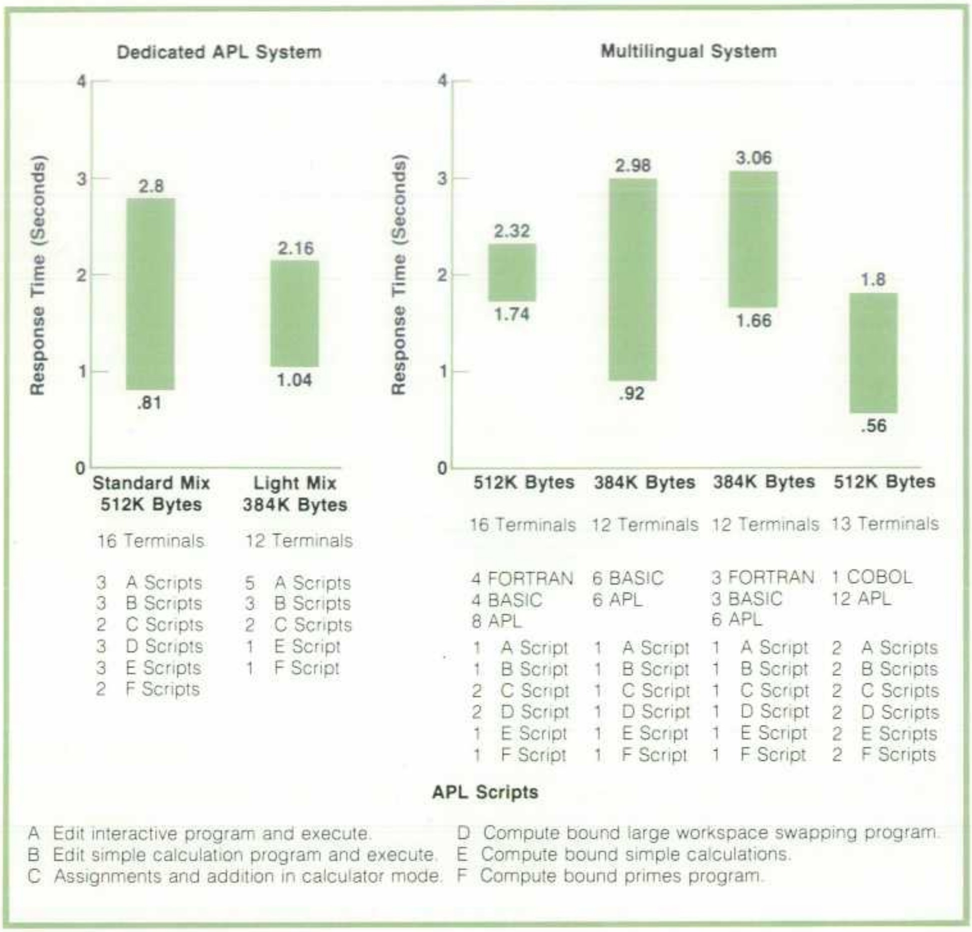


Fig. 1. *Average response times for a range of activities on an HP 3000 Series II System used only for APL and similar data for a range of APL activities on a multilingual HP 3000 Series II System. A system with 512K bytes of main memory will support up to 16 APL terminals.*

problem or application. Most other APL systems limit a workspace to 100,000 bytes or less. APL\3000 eliminates this limitation by giving each user a *virtual workspace*. A workspace is limited only by the amount of on-line disc storage available.

APL\3000 is the first APL system to include *APLGOL*[4] as an integral part. APLGOL is a block-structured language that uses keywords to control the program flow between APL statements. To facilitate the editing of APLGOL programs, and to provide an enhanced style of editing for APL programs and user data, a new *editor* was added to the APL system. This editor can be used on both programs and character data, and includes many features never available before in APL.

One of the features of APL that makes program development easier is that program debugging can be done interactively. When an error is encountered in an APL program, an error message is displayed along with a pointer to the place where the error was detected. Execution is suspended at this point, and control is returned to the user. In other versions of APL, the user is allowed to reference or change only the variables that are accessible within the function in which the error occurred, and must resume execution within that function. APL\3000 has implemented a set of *extended control functions* that allow the user to access or change any variable in the workspace and resume execution within any function that has not yet completed execution. These extended control functions can be used to implement advanced programming techniques that were previously difficult or impossible to implement in APL. An example is backtracking, which involves saving the control state at various points in the computation and returning to a previously saved control state when an error is detected.

The new features of APL\3000 are described in detail in the articles that follow.

Performance Data

An HP 3000 Series II System with 512K bytes of main memory will support a maximum of 16 terminals using APL, or a combination of terminals using APL and other languages. Fig. 1 shows typical response times for various combinations of terminal types, APL program loads, and memory sizes.

Acknowledgments

The authors wish to acknowledge the contributions of John Walters and Rob Kelley, without whose efforts APL\3000 would never have become a product. John served as project leader during the development stage and was responsible for many of the technological

innovations that are included in the final product. Rob participated in the design of the incremental compiler and his expertise in APLGOL helped make this facility an integral part of APL\3000.

Many people contributed to the initial discussions that led to the design of the incremental compiler. In particular, Dick Sites was most responsible for sketching out the compiling techniques. Larry Breed and Phil Abrams helped us develop new techniques for compiling APL while maintaining compatibility with the original philosophy of APL. Jeff Mischkinsky was responsible for the implementation of APLGOL and the design of the APL\3000 editor. Alan Marcum offered design suggestions from a user's point of view that helped us refine the product.

Our special thanks must go to Jim Duley, Paul Stoft, and Ed McCracken, whose long-standing support of our efforts helped us transform our ideas from a research project into a product.

References

1. K.E. Iverson, "A Programming Language," John Wiley and Sons, New York, 1967.
2. L.E. Shar, "Series II General-Purpose Computer Systems," Hewlett-Packard Journal, August 1976.
3. P.S. Abrams, "An APL Machine," PhD dissertation, SLAC Report No. 114, Stanford University, February 1970.
4. R.A. Kelley and J.R. Walters, "APLGOL-2, a Structured Programming System for APL," IBM Palo Alto Scientific Center, Technical Report G320-3318, 1973.

Kenneth A. Van Bree

Ken Van Bree received his bachelor's degree in electrical engineering from the University of Michigan in 1967, his master's degree from Massachusetts Institute of Technology in 1969, and the degree of Electrical Engineer, also from MIT, in 1971. During the summer of 1970 he helped develop a computer-aided design program for the HP 2100A Computer. Since joining HP Laboratories full-time in 1971, he's done computer-aided device modeling and mask layout for a 4K RAM, and helped design and implement the APL\3000 compiler. He's a member of IEEE. Ken was born in Newark, New Jersey and grew up in the state of Michigan. He's single, lives in Mountain View, California, and enjoys backpacking, scuba diving, motorcycles, photography, gourmet cooking, and designing and building his own furniture.

Introduction to APL

APL (an abbreviation for A Programming Language) is a concise high-level language noted for its rich variety of built-in (primitive) functions and operators, each represented by a symbol, and its exceptional facility for manipulating arrays.

APL uses powerful symbols in shorthand fashion to define complete functions in very few statements or characters. For example, the sums of each of the rows in a very large table called T are +/T. The sums of the columns are +/[1]T. The grand total of all numbers in the table is simply +/,T. Sorting and adding tables and other common operations are just as simple.

These characteristics, combined with minimal data declaration or other language requirements, help substantially reduce programming effort. Typical interactive APL programs take only 10-30% as long to write as would equivalent programs in other languages, such as FORTRAN or BASIC.

APL was invented by Dr. Kenneth E. Iverson at Harvard University. In 1962 a description of his mathematical notation was published. By 1966, IBM had refined the notation into a language and implemented the first version of APL on an experimental timesharing system. By 1969 APL was an IBM program product and several independent timesharing services began providing it.

Because APL is both easy to use and tremendously powerful it has gained widespread acceptance. A large, swiftly growing APL timesharing industry has developed. Approximately 70% of IBM's internal timesharing is done in APL. Over 50 North American universities including Yale, MIT, UCLA, Syracuse, University of Massachusetts (Amherst), York, and Wharton have in-house systems. Popularity has grown in Europe, especially Scandanavia and France.

Although initially designed for scientific environments, APL's features proved to be ideal for processing business data in tabular formats. Now, most timesharing services find approximately 80% of their APL business is in the commercial applications area.

APL Characteristics

A symbolic language with a large number of powerful primitive functions.

Uses right to left hierarchy (as opposed to precedence) that can be overridden by parentheses.

Designed to deal with arrays of numbers as easily as other languages deal with individual items.

Minimum language constraints: very few syntax rules; uniform rules for all data types and representations; automatic management of data storage and representation.

APL Advantages

Programs can be developed in 10-30% of the time and code space required by languages like FORTRAN, ALGOL, and BASIC.

Concepts of a program can often be more quickly grasped because of the brevity and conciseness of APL code.

Very flexible: programs easy to change; data very accessible and easy to rearrange.

Fig. 1. *Characteristics and advantages of APL*

BASIC

```
10 DIM A(100)
20 READ N
30 S=0
40 FOR I=1 TO N
50 READ A(I)
60 S=S+A(I)
70 NEXT I
80 PRINT S
90 END
```

FORTRAN

```
   DIMENSION A (100)
   READ (5,10) N
10 FORMAT (13)
   READ (5,20) (A(I), I=1, N)
20 FORMAT (8E10.3)
   S=0.0
   DO 30 I=1, N
30 S=S+A(I)
   WRITE (6,40)S
40 FORMAT (E12.3)
   END
```

ALGOL

```
REAL S;
INTEGER I, N;
GET N;
BEGIN
  REAL ARRAY A (1:N);
  S:=0.0;
  FOR I:=1 TO N DO
    BEGIN
      GET A(I);
      S:=S+A(I);
    END;
  PUT S;
END;
```

APL

```
+/,A←⎕
```

Fig. 2. *Comparison of steps required to read and sum a list of numbers.*

Given:

R = Revenues by product and salesman

	Johnver	Vanston	Danbree	Vansey	Mundyke
Tea	190	140	1926	14	143
Coffee	325	19	293	1491	162
Water	682	14	852	56	659
Milk	829	140	609	120	87

E = Expenses by product and salesman

	Johnver	Vanston	Danbree	Vansey	Mundyke
Tea	120	65	890	54	430
Coffee	300	10	23	802	235
Water	50	299	1290	12	145
Milk	67	254	89	129	76

Find:

Find each salesman's total commission where the formula for commission is 6.2% of profit, no commission for any product to total less than zero.

Answer:

	Johnver	Vanston	Danbree	Vansey	Mundyke
Commission	92	5	113	45	32

Explanation of APL Code Required:

```
.062 × +⌿ 0⌈ R−E
```

Step 1. Subtract each item in matrix E from each item in matrix R

Step 2. Find maximum of each item in resultant matrix versus the value of zero

Step 3. Sum over new resultant matrix by rows

Step 4. Multiply individual items in resultant vector by .062

Step 5. Automatically print new resultant vector

Comparison of APL Code Required With BASIC Code Required:

APL

```
.062 × +⌿ 0⌈ R−E
```

BASIC

```
 10 FILES DATA
 20 DIMENSION R(4,5), E(4,5), T(5)
 30 MAT READ #1; R,E
 40 MAT T = ZER
 50 FOR P = 1 TO 4
 60 FOR S = 1 TO 5
 70    T(S) = T(S) + .062*(R(P,S)−E(P,S)) MAX 0
 80 NEXT S
 90 NEXT P
100 MAT PRINT T
110 END
```

Fig. 3. *Explanation of APL code using typical example*

APL Data: Virtual Workspaces and Shared Storage

by Grant J. Munsey

MUCH OF THE POPULARITY of APL can be attributed to the convenient way it handles data. Most other programming languages treat variables as volatile "scratchpad" areas that are occupied by meaningful data only while programs are executing. Before programs can run, they must load the variables with data, usually by reading a file. During program execution the data is accessed by referring to variable names. When execution is completed, the meanings of the variables are lost unless the programs explicitly save their data in another file. APL, on the other hand, provides direct access to named data items, large or small, without forcing the concept of a file on the programmer. Once values are assigned to APL variables, they are accessible by name either in program execution mode or in calculator mode. The relationship between the data and the name is preserved until the programmer chooses to purge the data. The variables and the functions that operate upon them are preserved together, which means that APL applications need not go to files to access and save data.

In APL a unique name is attached to each distinct set of data by means of the assignment arrow:

```
DATE←7 4 1776
OCCASION← 'INDEPENDENCE DAY'
```

APL\3000 variables may be either scalar (single-element) or array-shaped with up to 63 dimensions. Though conceptually there are only two data types in APL, character and numeric, APL\3000 actually stores its data in a variety of ways for efficiency. APL differs from most other programming languages in that an APL programmer is never involved in specifying or choosing these machine-dependent internal representations; the APL system automatically chooses both the most efficient and the most accurate representation for any given set of data.

Likewise, an APL programmer never writes declarations specifying the shape, size, or amount of storage that will be required for a variable. Variables are declared by assigning data to them, and the APL system allocates the appropriate storage in which to retain the data. Readers familiar with languages requiring declaration of variables (e.g., FORTRAN, BASIC, COBOL) will recognize that the task of setting up such declarations can often take a substantial amount of programming time.

An interesting and useful feature of APL is that a particular variable name may, at different points in time, refer to different types and shapes of data, as the following sequence illustrates:

```
    A←3.5
    A←2 4 6 8
    A←'WHAT WOULD WE APPRECIATE?'
    A←2 3ρ1 2 3 4 5 6
    A
1 2 3
4 5 6
```

In this example, A is first assigned the numeric scalar 3.5. Then A is assigned the numeric vector 2 4 6 8. Next, A is assigned the character vector 'WHAT WOULD WE APPRECIATE?'. Finally, A is assigned a two-row, three-column array of numbers, then printed. Notice that each statement whose result is not explicitly assigned causes the result to be automatically printed.

The Workspace Concept

As functions and data are created, they remain associated with their user-assigned names in an area called the active workspace. This area can be named and saved for later use by entering the system command:

)SAVE *WSID*

where *WSID* is a user-specified workspace name. This saves a "snapshot" of all currently defined functions and data items. A saved workspace may be later reactivated by entering the system command:

)LOAD *WSID*

The concept of workspaces provides a convenient means for working on several different problems, each of which has its own set of pertinent data. For example, an accountant might have several customers for whom he is keeping payrolls. Several workspaces might be maintained, each containing payroll information for a particular client. Whenever a salary report is needed for a client, the appropriate workspace could simply be loaded and the report generated. Notice that workspaces are much like folders in a filing system; each holds the information required for a specific job.

Since all functions and data for a problem are stored in a single workspace, workspaces tend to grow very large as problem size increases. Yet most existing APL implementations have limited the size of workspaces, typically to less than 100,000 bytes. This constraint either imposes an artificial limit on the size of applications attempted, or forces the more determined programmer to seek additional storage outside of the workspace by explicit use of a file system, a definite violation of the general spirit of APL programming.

The HP 3000 is a small computer with a limited amount of main storage. Yet APL\3000 has avoided the traditional workspace size restrictions by employing two strategies: shared data storage and virtual workspaces.

Shared Data Storage

Shared data storage helps solve the workspace size problem by conserving storage. Multiple copies of the same data are avoided in many cases by allowing arbitrary numbers of variables to share the same data area. Consider the following two statements:

```
A← 1 2 5 6 9 10
B←A
```

The first statement creates a data area for A, while the second specifies that B is to be assigned whatever is in A. While one could naively make a second copy of the data and attach it to B, this is completely unnecessary and is a waste of storage; B should be able to share the original data with A.

A potential problem is: if A and B share the same data area, what happens if either of the variables changes part of its values? Does this affect the other variable? For instance, the subscripted assignment

```
B[3]←20
```

should not have the effect of also making A[3]'s value 20.

Copy-on-Write

APL\3000 solves this problem by attaching a reference count to every data area, and keeping track of how many variables are referring to it. Partial changes to a data area (e.g., B[3]←20) are allowed only if its reference count is 1 (i.e., it is unshared). A data area whose reference count is greater than 1 is never changed, since more than one variable is referring to it. Instead, a "copy-on-write" policy is adopted: the variable to be written into is given its own private copy of the data, the reference count of the original shared data area is decreased by 1, and the original data remains unchanged.

Shared data storage is useful in that it frequently allows the APL system to avoid making multiple copies of identical data. But this is really only a welcomed side effect of the real purpose of shared storage: allowing the dynamic compiler to implement certain selection functions and operators by applying Abrams' *subscript calculus*.[2] This technique is used to improve the performance of APL\3000, providing a two-fold justification of shared storage: space and speed.

Subscript calculus places another requirement on the APL system besides shared data areas: a variable's data area must be decoupled from its accessing information. That is, the data area itself must not describe the method of storing the data therein. To understand why this is required to perform subscript calculus, the attributes of APL data must be recalled: it has some actual collection of values, and it has a particular size and shape. Consider a numeric variable ABC whose data is arranged in two rows and three columns:

```
    ABC
1 2 4
0 5 9
```

The storage for ABC contains six data elements that the user thinks of as a two-dimensional array. At the machine level, however, storage is actually accessed in a linear fashion, as if it were a vector. To access any given element of ABC, the APL system takes a set of user indexes, consisting of a number for each dimension in ABC, and calculates a linear address into the data area holding ABC's values.

It has been common practice to store data in what is called row major order. In this scheme, data is stored with the rightmost subscript varying the fastest. For example, the actual linear layout of the variable ABC stored in this order would be:

1	2	4	0	5	9
ABC[0;0]	ABC[0;1]	ABC[0;2]	ABC[1;0]	ABC[1;1]	ABC[1;2]

Notice that zero-origin indexing was used (the first element in any dimension is index 0). Zero origin will be used in all formulas and examples hereafter.

When data is stored in row major order, one can map a set of user indexes into a machine address by employing the formula:[1]

$$\text{ADDRESS} = \sum_{J=0}^{\text{RANK}-1} \text{I}[\text{J}] \times \prod_{K=J+1}^{\text{RANK}-1} \text{SHAPE}[\text{K}] \qquad (1)$$

where I is the set of user indexes. In addition to the user indexes, this formula requires some information about the data's exact size and shape: RANK is the number of dimensions in the array, and SHAPE is a vector of the sizes of each of the dimensions. Together RANK and SHAPE make up the variable's row major access information.

Applying equation 1 to calculate the actual address of the element ABC [0;2]:

```
I       : 0 2
RANK    : 2
SHAPE   : 2 3
ADDRESS = (I[0] × SHAPE [1]) + (I[1] × 1)
        = (0 × 3) + (2 × 1)
        = 2
```

Referring back to the description of how ABC is stored, it can be seen that ABC [0;2] is indeed at location 2. Thus for data stored in row major order, all that is needed to calculate the actual storage address of an array element from a set of user indexes is the RANK and SHAPE of the data.

In APL systems not concerned with performing subscript calculus, this accessing information is traditionally stored with the data itself, which makes every data area self-describing. Subscript calculus, on the other hand, wants to view data in many different ways without physically rearranging it. The operation of subscripting (e.g., ABC [1;1]), and the functions TAKE, DROP, REVERSAL, TRANSPOSE, and RESHAPE can be implemented so that they rearrange data without actually moving or copying it, but only if the data area's accessing information is not an integral part of the data. Consider, for example, the APL function that reverses the columns of an array.

```
      ABC
1 2 4
0 5 9
      RABC←⌽ABC
      RABC
4 2 1
9 5 0
```

If the result of the reversal must always be stored in row major order, then nothing can be done except to make a second copy of ABC's data for RABC, with its order rearranged. But if one can depart from row major storage order in this case, one can generate new access information for RABC, and it can share ABC's data area with no data movement required. This requires generalizing the storage mapping function developed above to allow other storage arrangements than row major. The new formula will be:

$$\text{ADDRESS} = \text{OFFSET} + \sum_{J=0}^{\text{RANK}-1} I[J] \times \text{DEL}[J] \qquad (2)$$

This generalized formula makes explicit something that equation 1 was able to imply by knowing that data was stored in row major order: OFFSET is always zero; and DEL [J] is always

$$\prod_{K=J+1}^{\text{RANK}-1} \text{SHAPE}[K].$$

The new formula requires that both of these be made part of a variable's data accessing information. Equation 2 can be checked by again calculating the address of element ABC [0;2]:

```
I        : 0 2
RANK     : 2
SHAPE    : 2 3
DEL      : 3 1
OFFSET   : 0
ADDRESS = OFFSET + (I[0] × DEL [0]) + (I[1] × DEL [1])
        = 0 + (0 × 3) + (2 × 1)
        = 2
```

This is the same address calculated by applying equation 1, so equation 2 seems to work, at least on row major data. This new formula can be used to share ABC's data with RABC:

1	2	4	0	5	9
ABC[0;0]	ABC[0;1]	ABC[0;2]	ABC[1;0]	ABC[1;1]	ABC[1;2]
RABC [0;2]	RABC [0;1]	RABC [0;0]	RABC [1;2]	RABC [1;1]	RABC [1;0]

By changing both the DEL vector and the OFFSET as shown below, RABC can be totally described by its accessing information. As a check, equation 2 can be used to calculate the storage address of element RABC [0;2]:

```
I        : 0 2
RANK     : 2
SHAPE    : 2 3
DEL      : 3 −1
OFFSET   : 2
ADDRESS = OFFSET + (I[0] × DEL [0]) + (I[1] × DEL [1])
        = 2 + (0 × 3) + (2 × (−1))
        = 0
```

Referring back to the data area shared by ABC and RABC, it can be seen that RABC [0;2] is indeed at address 0 of the shared data area.

Thus by including the DEL vector and the OFFSET in a variable's set of accessing information, data areas can be shared among variables whose conceptual orderings differ. Notice, though, that each variable must have its own private set of accessing information for this to work, otherwise the shared data area can only be interpreted as one shape and storage method. Using a set of transformations to the DEL vector and the OFFSET in the above manner to rearrange data without actually moving it is the essence of subscript calculus.

Virtual Workspaces

The WS FULL message is well-known to most APL programmers. In specific terms, it means that the active workspace has filled up and program execution has stopped. In more general terms, it usually means that the programmer is going to have to do a lot of work to circumvent the problems of limited workspace size.

APL\3000 uses a technique called virtual storage to remove the workspace size limit. This allows the user to create and maintain workspaces containing millions of bytes of data. In fact, workspaces are limited in size only by the amount of disc storage available on the machine, the same limit that would apply to data stored explicitly as files.

Two layers of virtual workspace implementation make this possible. The first layer creates, by means of microcode routines, a very large linearly addressed data space. The second layer maintains this address space in many smaller variable-length segments.

To provide the large address space required to support virtual workspaces, APL\3000 uses a set of nine

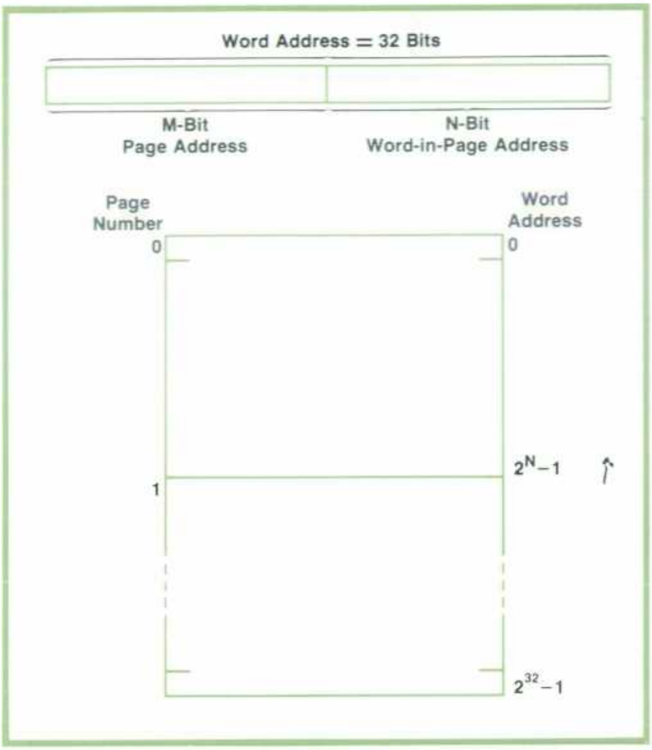


Fig. 1. *APL\3000 uses a virtual memory scheme to give each user whatever size workspace is needed, instead of imposing a fixed maximum workspace size as most APL systems do. The virtual memory is partitioned into 2^M pages of 2^N words each where $N+M = 32$.*

virtual memory instructions that have been added to the HP 3000 Series II instruction set. These instructions are added by installing eight read-only memory (ROM) integrated circuits in the CPU when APL\3000 is installed. The virtual memory instructions take a small amount of main computer storage plus a large disc file and create what looks like one large linearly addressed memory. This is done using what is known as a least recently used (LRU) virtual memory scheme.

The logical addresses used by APL\3000 are 32-bit quantities. The M most significant bits of the address are considered the *page address* and the N least significant bits the *word-in-page* address. Thus the virtual memory is partitioned into 2^M pages of 2^N words each (see Fig. 1). The values for N and M are determined by APL\3000 to provide efficient use of the computer hardware. N plus M must add up to 32, so the virtual memory can contain up to 2^{32} words (4,294,967,296 words). This is the only theoretical limit to workspace size.

The HP 3000 main computer store is set up to contain a number of 2^N-word pages from the virtual memory along with a small status table for each main-store-resident page. Each status table contains the following information:

- The virtual memory address of the first word in the page
- A link that points to the next status table
- An indicator that tells whether data in the page has been modified since the page was brought into main storage from the disc
- The main storage address of the words in the page.

Fig. 2 shows how these status tables are arranged in main store along with the data from the pages. The status tables are arranged in a list with each status table pointing to the next status table. This list is always arranged so the status table for the most recently used page is the first entry in the list.

Operation of the virtual memory instructions can be illustrated by describing the execution of a VIRTUAL LOAD instruction (see Fig. 3). This instruction requires a 32-bit virtual address as its operand and returns the word stored at that location in virtual memory. To accomplish this the first task is to determine the page in which the word resides (the page address). This is done by taking the M most significant bits of the virtual memory address. The second operation is to find where the required page resides. This is done by first searching down the list of status tables to see if the page is in main storage. If the page is found in the list then the word requested is already in main storage and all that need be done is to use the

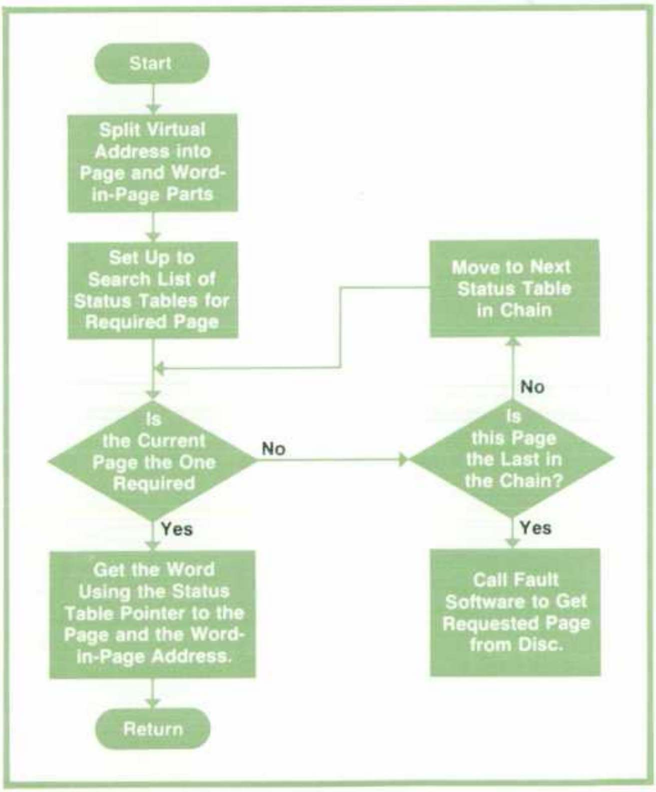


Fig. 2. *At a given time, the main computer store contains a number of 2^N-word pages from the virtual memory along with a small status table for each of these pages. The status tables are arranged in a list with the table for the most recently accessed page at the top of the list.*

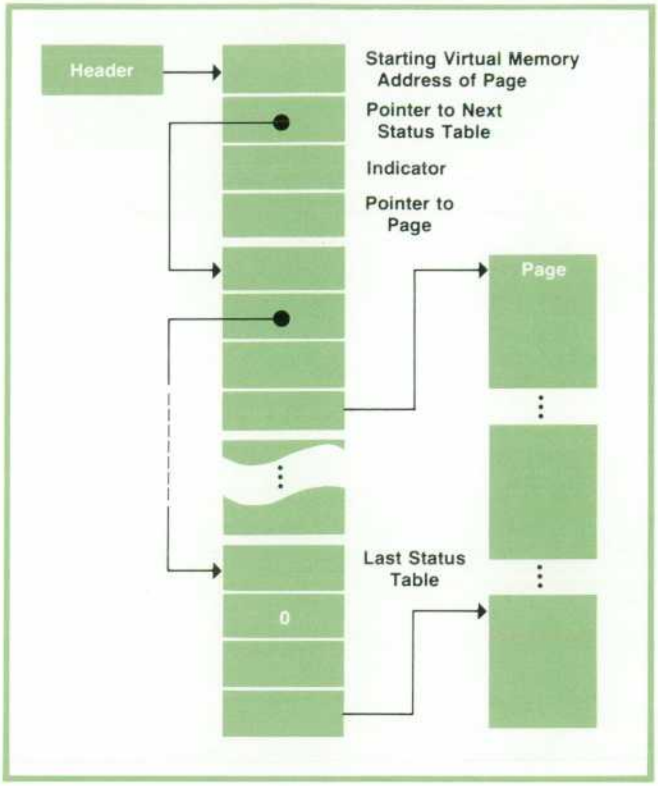


Fig. 3. *If the page that contains the word addressed is not in main storage, the system brings in the required page from the disc, swapping it for the page whose status table is at the bottom of the list, that is, the least recently used page.*

word-in-page part of the virtual address to access it. If the end of the status table list is reached without encountering the required page then a software routine is called from the virtual instruction microcode. This routine decides which of the current main-store-resident pages can be overwritten with the data from the new page, stores the current page on the disc if it has been altered since being loaded, and reads in the new page.

APL\3000 always chooses the least recently used page as the one that can be removed. This is the page whose status table is the last one in the status table list, since the list is maintained with the most recently used page first. This method is critical to the efficient operation of virtual memory, because it causes the pages that are used frequently by APL\3000 to remain in main storage where they can be rapidly accessed while the infrequently used pages migrate to the disc.

Virtual Segmentation

For this large linearly addressable virtual memory to be useful in creating virtual workspaces the address space must be broken up into several small blocks of memory, each of which can be independently expanded or contracted in size. In APL\3000 this is accomplished by three software routines. The first routine allocates blocks of memory; it is given the required number of words and it returns the starting virtual address of the block allocated. The second routine returns previously allocated blocks of memory to the free list where they are available for later reallocation. The third routine can be instructed to expand or contract the size of a currently allocated block of memory.

The virtual storage allocation routines work with a data structure called the free storage list (FSL). The FSL contains an entry for each block of unused storage in the virtual workspace. Each entry in the FSL contains the following items:

- A 32-bit virtual memory address that is the beginning of a free block of virtual memory
- The number of words in the free block of memory.

When a block of storage is returned to the FSL by the software a description of the block is put into the FSL so that no two FSL entries describe adjacent areas of memory. In this way the free storage available in a workspace is represented by the minimum number of FSL entries.

Conclusion

APL is a convenient language because its workspace concept allows the programmer to use variables rather than files. APL\3000 has extended its usefulness by allowing workspaces to be extremely large. Also, storage use and speed have been optimized by means of shared data areas and subscript calculus.

Acknowledgments

Mention should be made of three people who contributed to the design of the data handling portion of APL\3000. Jim Duley produced some of the initial ideas for the virtual memory system, John Sell worked day and night to get the microcode running, and Doug Jeung helped tune the code for the HP 3000.

References

1. D.E. Knuth, "The Art of Computer Programming, Vol. I;" Addison Wesley, 1968.
2. P.S. Abrams, "An APL Machine," PhD dissertation, SLAC Report No. 114, Stanford University, February 1970.

Grant J. Munsey

Grant Munsey was born in Los Angeles and attended the nearby University of California at Irvine, graduating in 1971 with a BSEE degree. For the next 3½ years he provided software support for HP's Neely Sales Region, then joined HP Laboratories to work in software research and development, mainly on APL. He's a member of ACM. Grant is single and lives in Sunnyvale, California. He's interested in aerobatic flying, photography, and sports cars.

APLGOL: Structured Programming Facilities for APL

by Ronald L. Johnston

OVER A PERIOD OF YEARS the computer science community has developed a set of programming disciplines for systematic program design that have become widely known as structured programming. One very important component of this science is a set of interstatement control structures for clearly expressing the flow of control. These control structures are embodied in such block-structured languages as ALGOL or PASCAL, and therefore these languages have been widely used in teaching computer science in colleges and universities. One control structure that has received much criticism as unstructured and harmful is the GOTO of FORTRAN and other languages.[1,7,8] The use of the GOTO, it is argued, is to be avoided because it can render program flow unintelligible, unmaintainable, and impossible to prove correct.

APL is a modern language with array-oriented functions, but only a single branching construct is available: →expression, where "expression," however complex, evaluates to a statement number to which control is transferred. This construct is the rough equivalent of a computed GOTO which, as mentioned previously, is not considered a good structured programming tool. Many APL enthusiasts, in defense of the language, have argued that its rich set of array functions reduces the necessity of including explicit loop constructs in an APL program, thereby minimizing the importance of good control structures in this particular language. Nevertheless, empirical studies[2] of APL programs have shown that the frequency of branching per line is greater in APL than in FORTRAN, although there are fewer branches per equivalent function. Furthermore, as a consequence of having only one branching construct the control flow even within well structured APL programs can often be obscure.

Many attempts have been made to improve the readability and understandability of the APL branch function. Saal and Weiss[2] relate that APL programmers use various stylized forms of branching with great frequency in an attempt to impart some regularity to the branch construct. These constructs have become much-used idioms of the language. Other APL programmers,[3,4,5] dissatisfied with even these stylized branching constructs, have invented special packages of APL functions that attempt to provide more acceptable control structures like IF-THEN-ELSE, WHILE-DO, CASE, and REPEAT-UNTIL. However, these special functions have discouraged their own use because they occupied storage in workspaces that were already too small, and because the function calls imposed a run-time speed penalty on the user. The only acceptable solution lay in enhancing the language itself, so that APL programmers could use the growing body of structured programming techniques without incurring the penalties inherent in the solutions to date.

Solution: APLGOL

APL\3000 includes an alternate language, APLGOL, which enhances standard APL in the area of branching. Based on the work of Kelley and Walters[6], APLGOL is a fully-supported language that adds ALGOL-like control structures to APL to provide the needed structured programming facilities. Programmers writing in APLGOL can make use of such familiar constructs as IF-THEN-ELSE, WHILE-DO, REPEAT-UNTIL, and CASE. Some constrained forms of structured branching are also included; they are LEAVE, ITERATE, and RESTART. The resultant programs are much easier to read, understand, and maintain than the equivalent programs written in standard APL. These qualities are essential in production programming environments.

Another language facility, ASSERT, has been incorporated to encourage programmers to assert correctness properties of algorithms as they write them, hopefully to foster the proof-of-correctness approach to programming that Dijkstra has recognized as so important to the production of error-free programs.[8,9,10] Using ASSERT statements the programmer states properties and conditions that must be true if the program being written is to work properly. For example, suppose a function uses the variable A as a divisor and the programmer expects that no element of A should ever be zero. The following assertion might be included in the function ahead of the division:

```
ASSERT 1: ∧/A≠0;
```

APL:

```
[ 0] LISTFNS;LISTFNS;FNL;FNAME;⎕IO;NLINES;INX
[ 1] ⍝ PRINTS TEXT OF ALL FUNCTIONS IN WORKSPACE EXCEPT ITSELF
[ 2] ⎕IO← 0
[ 3] FNL←FNL[⍋65↑⍉⎕AV⍳FNL←⎕NL 3 4;] ⍝ GET SORTED FNS LIST
[ 4] →(0≠×/⍴FNL)/HAVEFNS
[ 5] ⎕←'(NO FUNCTIONS IN WORKSPACE)'
[ 6] →0
[ 7] HAVEFNS: ⎕←'          )FNS',⎕R,,FNL,'  '⍝ PRINT )FNS LIST
[ 8] INX←0
[ 9] NEXTNAME:   →(INX≥1↑⍴FNL)/0
[10] FNAME←(FNAME≠' ')/FNAME←,FNL[INX;] ⍝ DE-BLANK NAME
[11] ⎕←(2⍴⎕R),'***** ',FNAME,' *****'
[12] NLINES←1↑⍴CR←⎕CR FNAME ⍝ GENERATE CANONICAL REP
[13] ⎕← '[',(((⌈10⍟NLINES),0)⍕(NLINES,1)⍴⍳NLINES),']',' ',CR
[14] INX←INX+1
[15] →NEXTNAME
```

APLGOL:

```
[ 0] PROCEDURE LISTFNS,LISTFNS,FNL,FNAME,⎕IO,NLINES,INX;
[ 1]    ⍝ PRINTS TEXT OF ALL FUNCTIONS IN WORKSPACE EXCEPT ITSELF ⍝
[ 2]    ⎕IO← 0;
[ 3]    FNL←FNL[⍋65↑⍉⎕AV⍳FNL←⎕NL 3 4;]; ⍝ GET SORTED FNS LIST ⍝
[ 4]    IF 0=×/⍴FNL THEN
[ 5]       ⎕←'(NO FUNCTIONS IN WORKSPACE)'
[ 6]    ELSE
[ 7]       BEGIN
[ 8]          ⎕←'          )FNS',⎕R,,FNL,'  '; ⍝ PRINT )FNS LIST ⍝
[ 9]          INX←0;
[10]          WHILE INX<1↑⍴FNL DO
[11]             BEGIN
[12]                FNAME←(FNAME≠' ')/FNAME←,FNL[INX;]; ⍝ DE-BLANK NAME ⍝
[13]                ⎕←(2⍴⎕R),'***** ',FNAME,' *****';
[14]                NLINES←1↑⍴CR←⎕CR FNAME; ⍝ GENERATE CANONICAL REP ⍝
[15]                ⎕←'[',(((⌈10⍟NLINES),0)⍕(NLINES,1)⍴⍳NLINES),']',' ',CR;
[16]                INX←INX+1;
[17]             END;
[18]       END;
[19] END PROCEDURE
```

Fig. 1. *An APL function and its APLGOL counterpart. The two functions are nearly identical, but the APLGOL function makes use of ALGOL-like control structures that make it easier to read, understand, and maintain.*

In this fashion the programmer lets the correctness proof and the program grow hand in hand. Each ASSERT statement contains a relational expression that is evaluated dynamically each time control reaches it. If the assertion proves false, execution is halted to permit the programmer to choose an appropriate course of action. Assertion statements can be conditionally executed, based on a level number in each assertion. One useful way to employ assertions is to have all assertions checked during initial program writing and debugging. Later, as the program reaches production status, assertion checking is turned off. If at some future date the program exhibits erroneous behavior, checking of assertions can be easily reinitiated, greatly facilitating debugging efforts. Using assertions in this fashion, there is no run-time penalty during production use of the programs; only during debugging stages are the assertions checked.

A workspace may contain both APL and APLGOL functions, which may call each other without restriction. (However, any given function must be entirely APL or entirely APLGOL.) APLGOL expressions are exactly the same as APL expressions, following the same set of syntax and semantic rules. A function originally developed in APL can be easily modified to become an APLGOL function, and vice versa. The only differences between APL and APLGOL functions lie in the specific syntax of the function headers, the control structures, the use of the lamp symbol (⍝) as a comment terminator, and the fact that APLGOL, like ALGOL, terminates statements with a semicolon. Fig. 1 contrasts an APL function with its equivalent APLGOL function, illustrating how nearly identical the two functions are.

Canonical Forms

For run-time efficiency, it has been customary for APL interpreters to translate functions from character form into an internal form, whereupon the original character source is discarded. Subsequent requests for display of the functions are satisfied by translating the internal form back to a canonical character form. APL programmers have become accustomed to this canonical form of their programs being slightly different from what they originally input, in that unnecessary blanks have been compressed out, labels "undented", and the formats of numeric constants perhaps changed. The short function shown below illustrates how the original and canonical forms may differ for APL:

Original APL

```
[0] R←    PART PERCENT    WHOLE
[1] R    ←1E2 × PART÷ WHOLE
```

Canonical APL

```
[0] R←PART PERCENT WHOLE
[1] R←100×PART÷WHOLE
```

In similar fashion, APLGOL translates to internal form and back-translates to a stylized canonical form. However, APLGOL canonical form may be markedly different from the original. APLGOL can be input free-form with many statements per line, but the canonical form always has one statement per line, with indenting for each layer of nesting. As Fig. 2 shows, the canonical form of this function offers the advantage of making the control structures more obvious by indenting the IF-THEN-ELSE statements.

One consequence of the APLGOL control structures is that the keywords of these structures (IF, THEN, etc.) are reserved and cannot be used as variable or function names in APLGOL functions. This is not usually a severe limitation to the programmer.

Important Design Considerations

APLGOL is a fully-supported language, not an add-on to APL. The decision was made early in the

```
Original APLGOL
[ 0]  PROCEDURE A CONFORMS B:IF (∨/1=(×/ρA),×/ρB)THEN
[ 1]  'CONFORMABLE – SCALAR/UNIT EXTENSION' ELSE IF (ρρA)=ρρB
[ 2]  THEN IF (ρA)∧.=ρB THEN'CONFORMABLE – SAME SHAPE' ELSE
[ 3]  'NOT CONFORMABLE – LENGTH ERROR'
[ 4]  ELSE 'NOT CONFORMABLE – RANK ERROR';
[ 5]  END PROCEDURE

Canonical APLGOL
[ 0]  PROCEDURE A CONFORMS B:
[ 1]    IF (∨/1=(×/ρA),×/ρB) THEN
[ 2]      'CONFORMABLE – SCALAR/UNIT EXTENSION'
[ 3]    ELSE
[ 4]      IF (ρρA)=ρρB THEN
[ 5]        IF (ρA)∧.=ρB THEN
[ 6]          'CONFORMABLE – SAME SHAPE'
[ 7]        ELSE
[ 8]          'NOT CONFORMABLE – LENGTH ERROR'
[ 9]      ELSE
[10]        'NOT CONFORMABLE – RANK ERROR';
[11]  END PROCEDURE
```

Fig. 2. *User inputs in APLGOL are translated to an internal form and back-translate to a canonical form. The canonical form makes the control structures more obvious by indenting.*

design stages that it was to be as convenient to use as APL and should require no extra steps for the programmer. It was to suffer no significant speed or space penalties, but should offer itself as a viable alternative to programming in APL.

One important design decision was to use the same dynamic incremental compiler for both APL and APLGOL (see article, page 17). Once a function has been translated to internal form (S-code), its incremental compilation and execution is handled by a single mechanism that is common to both languages. The most obvious payoff from this approach is that only one such system needed to be implemented, resulting in lower development costs than if two separate compilers had been written. A second, less obvious advantage is that this guarantees that there are no insidious semantic differences in the way each language evaluates its expressions. That is, an expression like +/'' gives the same result (DOMAIN ERROR in some systems, including ours; 0 in other systems) in both languages. Finally, it guarantees that the execution speed of both languages is the same, except in functions dominated by branching overhead. In these cases APLGOL tends to be slightly faster, because it generates more efficient branching code. APLGOL branches don't have to be range-checked at run time as APL branches do, since all APLGOL branches are generated and guaranteed in-range by the character-to-internal translator when the function is created.

These considerations continually influenced the design of APL\3000, most often having the effect of complicating internal code assignments, data structures, and support routines. The result, however, is a system that honestly supports both APL and APLGOL without noticable favoritism of either.

References

1. E.W. Dijkstra, "GOTO Statement Considered Harmful," Communications of the ACM, 11 (1968), pp. 147-148.
2. H.J. Saal and S. Weiss, "An Empirical Study of APL Programs," IBM Israel Scientific Center, Technion City, Haifa, Israel.
3. J.P. Dorocak, "APL Functions which Enhance APL Branching," IBM Corp., Federal Systems Division, Oswego, New York, APL 76 Proceedings (1976), pp. 99-105.
4. W.K. Giloi and R. Hoffman, "Adding a Modern Control Structure to APL without Changing the Syntax," APL 76 Proceedings (1976), pp. 189-194.
5. L.R. Harris, "A Logical Control Structure for APL," APL Congress 1973, American Elsavier, New York, 1973, pp. 203-210.
6. R.A. Kelley and J.R. Walters, "APLGOL-2, A Structured Programming System for APL," IBM Palo Alto Scientific Center, Technical Report No. G320-3318, 1973.
7. "The GOTO Controversy," SIGPLAN Notices (Special Issue on Control Structures in Programming Languages), Vol. 7, No. 11, 1972.
8. E.W. Dijkstra, "The Humble Programmer," 1972 Turing Lecture, Communications of the ACM, Vol. 15 No. 10, October 1972.
9. E.W. Dijkstra, O.J. Dahl, and C.A.R. Hoare, "Structured Programming," Academic Press, London, October 1972.
10. R.W. Floyd, "Assigning Meanings to Programs," Proceedings of Symposium on Applied Mathematics, American Mathematical Society, Vol. 19, 1967, pp 19-32.

Ronald L. Johnston

Ron Johnston graduated from the University of California at Santa Barbara in 1973 with a BS degree in electrical engineering and computer science. He joined HP Laboratories that same year, designed a CRT-based interactive text editor, and then helped design and implement APL\3000. He's now APL project manager. A native of Southern California, Ron is married, has a two-year-old daughter, and lives in Sunnyvale, California. Besides APL, Ron's passions are off-road motorcycling and music—he plays guitar and sings in a duo, the other half of which is his wife. He also serves as counselor for a church youth group and as tour director for a youth choir.

APL\3000 Summary

Primitive Functions and Operators

Monadic		Dyadic
IDENTITY	+	ADDITION
NEGATE	−	SUBTRACTION
SIGNUM	×	MULTIPLICATION
RECIPROCAL	÷	DIVISION
EXPONENTIAL	⋆	POWER
NATURAL LOGARITHM	⍟	GENERAL LOGARITHM
	∧	AND
	∨	OR
	⍲	NAND
	⍱	NOR
NOT	~	
	<	LESS THAN
	≤	LESS THAN OR EQUAL
	=	EQUAL
	≠	NOT EQUAL
	≥	GREATER THAN OR EQUAL
	>	GREATER THAN
ROLL	?	DEAL
PI TIMES	○	CIRCULAR FUNCTIONS
CEILING	⌈	MAXIMUM
FLOOR	⌊	MINIMUM
ABSOLUTE VALUE	∣	RESIDUE
FACTORIAL	!	BINOMIAL
SHAPE	⍴	RESHAPE
	↑	TAKE
	↓	DROP
AXIS (AUXILIARY)	[ ]	INDEXING
INDEX GENERATOR	⍳	INDEX OF
REVERSE	⌽ or ⊖	ROTATE
TRANSPOSE	⍉	GENERAL TRANSPOSE
REDUCTION	/ or ⌿	COMPRESSION
SCAN	\ or ⍀	EXPANSION
RAVEL	,	CATENATE
GRADE UP	⍋	
GRADE DOWN	⍒	
	∊	MEMBERSHIP
	⊥	DECODE
	⊤	ENCODE
EXECUTE	⍎	EXTENDED EXECUTE
FORMAT	⍕	EXTENDED FORMAT
MATRIX INVERSE	⌹	MATRIX DIVISION
	.	GENERALIZED INNER PRODUCT
	∘.	GENERALIZED OUTER PRODUCT

Overstrike Characters

Formed by striking one key, backspacing, striking other key. Order Immaterial.

Symbol	Made with	Symbol	Made with
⍟	○ ⋆	!	' .
⌽	○ |	⍝	∩ ∘
⊖	○ −	⍲	∧ ~
⍉	○ \	⍱	∨ ~
⌿	/ −	⍋	∆ |
⍀	\ −	⍒	∇ |
⍎	⊥ ∘	A̲	A _
⍕	⊤ ∘	⁝	: .
⌹	⎕ ÷	Z̲	Z _
⍞	⎕ '	⍙	∆ _

System Variables

⎕A	Alphabet Characters	⎕LX	Latent Expression
⎕AI	Account Information	⎕N	Null Character
⎕AL	APLGOL Assertion Level	⎕PP	Print Precision
⎕AV	Atomic Vector	⎕PW	Print Width
⎕B	Backspace Character	⎕R	Carriage Return Character
⎕CT	Comparison Tolerance	⎕RL	Random Link (Seed)
⎕D	Digit Characters	⎕SN	Stack Names
⎕E	Escape Character	⎕T	Horizontal Tab Character
⎕HT	Horizontal Tab Positions	⎕TT	Terminal Type
⎕IO	Index Origin	⎕TS	Time Stamp
⎕L	Line-Feed Character	⎕VM	Virtual Memory Characteristics
⎕LA	Language	⎕WA	Workspace Area Used
⎕LC	Line Counter	⎕WI	Workspace Identification

System Functions

Result		Syntax		Name
CM ←		⎕CR	*CV*	Canonical Representation
NS ←	{*CV*}	⎕CSE	*NV*	Capture Stack Environment
AV ←	*NSV*	⎕CV	*AV*	Convert
NS ←		⎕DL	*NS*	Delay
BV ←		⎕EX	*CVM*	Expunge
CV ←		⎕FX	*CVM*	Function Establishment (Fix)
NM ←		⎕MV	*CV*	Monitor Values
NV ←		⎕NC	*CVM*	Name Classification
CM ←	{*CV*}	⎕NL	*NSV*	Name List
BV ←		⎕QM	*CV*	Query Monitor
BV ←		⎕QS	*CV*	Query Stop
BV ←		⎕QT	*CV*	Query Trace
NV ←		⎕RSE	*NV*	Release Stack Environment
NV ←	{*NV*}	⎕RM	*CV*	Reset Monitor
NV ←	{*NV*}	⎕RS	*CV*	Reset Stop
NV ←	{*NV*}	⎕RT	*CV*	Reset Trace
NV ←	{*NV*}	⎕SM	*CV*	Set Monitor
NV ←	{*NV*}	⎕SS	*CV*	Set Stop
NV ←	{*NV*}	⎕ST	*CV*	Set Trace
NM ←	{*BVM*}	⎕SVC	*CVM*	Shared Variable Control
NV ←	{*CVM*}	⎕SVO	*CVM*	Shared Variable Offer
NV ←		⎕SVR	*CVM*	Shared Variable Retract
CM ←		⎕SVQ	*CV*	Shared Variable Query
CV ←		⎕VR	*CV*	Vector Representation

Notes:

AV:	Arbitrary Vector	*BM*:	Boolean Matrix
BV:	Boolean Vector	*CM*:	Character Matrix
CV:	Character Vector	*CVM*:	Character Vector or Matrix
NM:	Numeric Matrix	*NS*:	Numeric Scalar
NSV:	Numeric Scalar or Vector	*NV*:	Numeric Vector
{α}:	α Is Optional		

Miscellaneous

¯	Negative Constant Indicator
'	Character Constant Delimiter
←	Assignment
→	Branch
⎕	Evaluated Input, Output
⍞	Literal Input, Prompting Output
( )	Grouping
◇	Statement Separator
;	Statement Separator(APLGOL), List Separator
:	Label Indicator
⍝	Comment Delimiter

APLGOL Control Structures

ASSERT *INTEGER EXPRESSION*: *BOOLEAN EXPRESSION*
BEGIN *STATEMENT LIST* END
CASE *INTEGER EXPRESSION* OF *INTEGER CONSTANT*
    BEGIN
        CASE LABEL: *STATEMENT*;
        CASE LABEL: *STATEMENT*;
        ⋮
        CASE LABEL: *STATEMENT*;
        {DEFAULT: *STATEMENT*;}
    END CASE
EXIT {*EXPRESSION*}
FOREVER DO *STATEMENT*
HALT {*EXPRESSION*}
IF *BOOLEAN EXPRESSION* DO *STATEMENT*
IF *BOOLEAN EXPRESSION* THEN *STATEMENT*
    ELSE *STATEMENT*
ITERATE: *CONTROL STRUCTURE NAME LIST*
LEAVE: *CONTROL STRUCTURE NAME LIST*
NULL
PROCEDURE *HEADER*: *STATEMENT LIST* END PROCEDURE
REPEAT *STATEMENT LIST* UNTIL *BOOLEAN EXPRESSION*
RESTART: *CONTROL STRUCTURE NAME LIST*
WHILE *BOOLEAN EXPRESSION* DO *STATEMENT*

Notes:

{α}: α Is Optional

CONTROL STRUCTURE NAME LIST: List of Control Structure Names among CASE, FOREVER, IF, PROCEDURE, REPEAT, or WHILE. E.g.: IF,CASE

HEADER: Standard APL Function Header, except that Local Variables Are Preceded by a Comma instead of a Semicolon.

STATEMENT: One of the Above Control Structures, or an APL Expression.

STATEMENT LIST: One or More Statements, Each Terminated by a Semicolon.

Comments Have the Form: ⍝ COMMENT TEXT⍝

Editor Commands

A{DD}	Allows Entry of New Text
B{RIEF}	Changes Messages to Brief Mode (Short)
C{HANGE}	Substitutes One String for Another
CO{PY}	Copies Text from One Location to Another
CU{RSOR}	Changes the Line Pointer
D{ELETE}	Deletes Lines in the Edit Text
DELT{A}	Changes the Line Increment
END	Exits Editor, Making Text into a Function
F{IND}	Locates a String in the Text
H{ELP}	Prints Information about Editor Commands
L{IST}	Prints Lines of Text
LOCK	Similar to END, but Locks the Function
MAT{RIX}	Exits Editor, Creating a Character Matrix
M{ODIFY}	Modifies the Contents of a Line
QUIT	Exits Editor, Discarding the Changes
R{EPLACE}	Replaces Lines of the Text
RES{EQUENCE}	Renumbers and Moves Text Lines
U{NDO}	Negates the Effects of the Last Commands
VEC{TOR}	Exits Editor, Creating a Character Vector
VER{BOSE}	Changes Messages to Verbose Mode (Long)

Note:

{α}: α Is Optional. Commands May Be Abbreviated.

System Commands

)BIND	Turns Binding Messages ON or OFF
)CLEAR	Obtains New, Clean Workspace (WS)
)CONTINUE	Leaves APL, Saving WS in Workspace CONTINUE
)COPY *WSID* {*NAME LIST*}	Obtains Part or All of a Stored WS
)DEPTH {*INTEGER*}	Sets the Execution Stack Size
)DROP *WSID*	Deletes a Stored WS
)EDIT {*OBJECT NAME*}	Enters Editor, Working on *OBJECT NAME*
)ERASE *NAME LIST*	Deletes Objects in *NAME LIST* from Active WS
)EXIT	Leaves APL
)FILES {*GROUP* {.*ACCOUNT*}}	Lists Stored Files
)FNS {*LETTER*}	Lists Functions in Active WS
)HELP {*COMMAND NAME*}	Prints Information about System Commands
)LANGUAGE {APL *OR* APLGOL}	Specifies Default Language Processor
)LIB {*GROUP* {.*ACCOUNT*}}	Lists Stored APL Workspaces
)LOAD *WSID*	Makes a Copy of a Stored WS the Active WS
)MPE	Break from APL to MPE Command Interpreter
)OFF	Leaves APL
)PCOPY *WSID* {*NAME LIST*}	Like COPY, but Doesn't Replace Objects
)RESET {*ENVIRONMENT NUMBER*}	Sets an Environment to the Empty Environment
)RESUME	Resumes Execution of Suspended Function
)SAVE {*WSID*}	Stores the Active Workspace
)SI {*ENVIRONMENT NUMBER*}	Prints the State Indicator
)SIV {*ENVIRONMENT NUMBER*}	Prints the State Indicator Stack, with Local Variables
)TERM {*TERMINAL TYPE*}	Sets the Terminal Type
)TERSE	Sets Messages to Terse Mode (Short)
)TIME	Turns Calculator Mode Timing ON/OFF
)VARS {*LETTER*}	Prints the Variables in the Active WS
)VERBOSE	Sets Messages to Verbose Mode (Long)
)WSID {*WSID*}	Changes the Active WS's Name

Notes:

{α}: α is optional

WSID: Workspace Identification

TERMINAL TYPE: One of AJ, ASCII, BP, CDI, CP, DM, GSI, or HP.

All Commands May Be Abbreviated.

Circular Functions

R ← A ○ B

A	R	A	R
¯7	arc tanh B	1	sin B
¯6	arc cosh B	2	cos B
¯5	arc sinh B	3	tan B
¯4	(¯1+B*2)*.5	4	(1+B*2)*.5
¯3	arc tan B	5	sinh B
¯2	arc cos B	6	cosh B
¯1	arc sin B	7	tanh B
0	(1−B*2)*.5		

SPECIFICATIONS AND FEATURES
APL\3000 (Language Subsystem 32105A)

APL\3000 is a language subsystem that runs under the control of Multiprogramming Executive (MPE) on the HP 3000 Series II Computer.

COMPATIBILITY: APLSV compatible, including system functions and variables, shared variable mechanism, Format (⍕), Execute (⍎), Scan(\), and Matrix Inversion and Division (⌹).

FILE SYSTEM: Full access to the Multiprogramming Executive (MPE) file system allows private or shared files via the Shared Variable mechanism, communication with other language subsystems, access to peripheral devices (line printers, card readers, magnetic tapes, discs, etc.).

APLGOL: An alternate language that provides modern ALGOL-like control structures in an APL environment. IF-THEN-ELSE, BEGIN-END, WHILE-DO, REPEAT-UNTIL, CASE, and ASSERT are among the constructs available.

EDITOR: Full function and text editing facilities are provided for by a powerful new editor. Includes features never before available to APL programmers, among them the ability to create and edit matrices and vectors. Provides such commands as CHANGE, COPY, FIND, RESEQUENCE, and UNDO, as well as a HELP facility for the novice or occasional user.

CONCEPTUAL DATA TYPES: Character and Numeric.

ACTUAL DATA TYPES: APL automatically chooses the appropriate internal representation for data from the following types:

CHARACTER: represented by 8-bit codes following the code assignments outlined by ⎕AV. Codes include lower-case ASCII alphabetics, control codes.

BIT: values 0 and 1 packed 16 per machine word for data of rank 1 (vector) or greater (array).

INTEGER: integer values within the range −32768 to 32767 are stored as 16-bit signed integers.

REAL: real values within the range $\pm(2^{-256}, 2^{+256})$ are stored as 64-bit floating point numbers. 16 decimal digit accuracy.

MAXIMUM ARRAY RANK: 63 dimensions.

MAXIMUM ARRAY SIZE: 32,767 elements.

ARITHMETIC PROGRESSION VECTORS: Integer vectors that can be described by the form A +B × ⍳C are stored as Arithmetic Progression Vectors (APV's), which require no data areas.

SHARED DATA AREAS: Variables of rank 1 (vector) or greater can share the same data areas, avoiding multiple copies of the same data. Shared data areas are duplicated only if one of the sharing variables attempts to change its data.

WORKSPACE SIZE: Limited only by the amount of on-line disc storage available. Initial size: 32,767 bytes. Automatically made larger as necessary. Practical limit: 400,000,000 bytes.

TERMINAL SUPPORT: Accepts terminals, with or without an APL character set, that use a standard ASCII interface at speeds from 110 to 2400 baud. Provisions made for both bit and character pairing terminals. Special support given to the HP 2641A Display Station to take advantage of its special features. The following other terminals have been tested: Anderson Jacobson 630, Computer Devices Teleterm 1030, Data Media Elite 1520, Gen-Com System Model 300.

ENVIRONMENT: Runs as a standard subsystem under control of Multiprogramming Executive (MPE). Allows batch APL jobs, simultaneous use of five other languages (BASIC, COBOL, FORTRAN, RPG, and SPL), networked access to other HP 3000's.

SYSTEM REQUIREMENTS AND PERFORMANCE: The minimum system required is an HP 3000 Series II with 256K bytes of memory operating under MPE II; for multilingual operation, at least 384K bytes of memory is needed. Operation with 10 or more terminals requires full memory (512K bytes). Maximum recommended number of simultaneous APL users is 16.

INSTALLATION: APL\3000 includes hardware microcode and must be installed by a factory authorized Customer Engineer. Installation is included in the list price.

ORDERING INFORMATION: 32105A APL\3000 Subsystem. Includes the dynamic compiler, hardware microcode, and the APL\3000 Reference Manual (32105-90002). All software supplied in object code form only.

PRICE IN U.S.A.: $15,000.

MANUFACTURING DIVISION: GENERAL SYSTEMS DIVISION
5303 Stevens Creek Boulevard
Santa Clara, California 95050 U.S.A.

SPECIFICATIONS
HP Model 2641A APL Display Station

General

SCREEN SIZE: 127 mm (5 in) × 254 mm (10 in)

SCREEN CAPACITY: 24 lines × 80 columns (1,920 character)

CHARACTER GENERATION: 7×9 enhanced dot matrix; 9×15 dot character cell; non-interlaced raster scan

CHARACTER SIZE: 2.46 mm (.097 in) × 3.175 mm (.125 in)

CHARACTER SET: 128 character APL; 64 character upper-case Roman; 64 character APL overstrike. (Note: the 2641A supports only one additional character set.)

CURSOR: Blinking underline

DISPLAY MODES: White on black; black on white (inverse video), blinking, half-bright, underline.

REFRESH RATE: 60 Hz (50 Hz optional)

TUBE PHOSPHOR: P4

IMPLOSION PROTECTION: Bonded implosion panel

MEMORY: MOS ROM: 24K bytes (program); RAM: std. 4096 bytes; 12 kilobytes max. (16K including max. data comm. buffer)

OPTION SLOTS: 5 available

KEYBOARD: Detachable, full APL/ASCII code bit-pairing keyboard, user-defined soft keys, and 18 additional control and editing keys; ten-key numeric pad; cursor pad; multispeed auto-repeat, N-key roll-over; 1.22m (4 foot) cable.

CARTRIDGE TAPE (option): Two mechanisms

READ/WRITE SPEED: 10 ips

SEARCH/REWIND SPEED: 60 ips

RECORDING: 800 bpi

MINI CARTRIDGE: 110 kilobyte capacity (maximum per cartridge)

Data Communications

DATA RATE: 110, 150, 300, 1200, 2400, 4800, 9600 baud, and external. Switch selectable. (110 selects two stop bits). Operating above 4800 baud in APL mode may require nulls or handshake protocol to insure data integrity.

STANDARD ASYNCHRONOUS COMMUNICATIONS INTERFACE: EIA standard RS232C; fully compatible with Bell 103A modems; compatible with Bell 202C/D/S/T modems. Choice of main channel or reverse channel line turnaround for half duplex operation.

OPTIONAL COMMUNICATIONS INTERFACES (see 13260A/B/C/D Communications data sheet for details):

Current loop, split speed, custom baud rates

Asynchronous Multipoint Communications

Synchronous Multipoint Communications - Bisync

TRANSMISSION MODES: Full or half duplex, asynchronous

OPERATING MODES: On-line; off-line; character, block

PARITY: Switch selectable; even, odd, none

Environmental Conditions

TEMPERATURE, FREE SPACE AMBIENT:

NON-OPERATING: −40 to +75°C (−40 to +167°F)

OPERATING: 0 to 55°C (+32 to +131°F)

TEMPERATURE, FREE SPACE AMBIENT (TAPE):

NON-OPERATING: −10 to 60°C (−15 to +140°F)

OPERATING: 5 to 40°C (+41 to 104°F)

HUMIDITY: 5 to 95% (non-condensing)

ALTITUDE:

NON-OPERATING: Sea level to 7620 metres (25,000 feet)

OPERATING: Sea level to 4572 metres (15,000 feet)

VIBRATION AND SHOCK (Type tested to qualify for normal shipping and handling in original shipping carton):

VIBRATION: 37 mm (0.015 in) pp, 10 to 55 Hz, 3 axes

SHOCK: 30g, 11ms, 1/2 sine

Physical Specifications

DISPLAY MONITOR WEIGHT: 19.6 kg (43 pounds)

KEYBOARD WEIGHT: 3.2 kg (7 lbs)

DISPLAY MONITOR DIMENSIONS: 444 mm W × 457 mm D × 342 mm H (17.5 in W × 18 in D × 13.5 in H).
648 mm D (25.5 in D) including keyboard.

KEYBOARD DIMENSIONS: 444 mm W × 216 mm D × 90 mm H (17.5 in W × 8.5 in D × 3.5 in H)

Power Requirements

INPUT VOLTAGE: 115 (+10% −23%) at 60 Hz (±0.2%)
230 (+10% −23%) at 50 Hz (±0.2%)

POWER CONSUMPTION: 85 W to 140 W max.

Product Safety

PRODUCT MEETS: UL requirements for EDP equipment, office appliances, teaching equipment; CSA requirements for EDP equipment; U.L. and CSA labels are applied to equipment shipped to the U.S. and Canada.

Ordering Example

Here is an example for ordering a 2641A Terminal with upper and lower case Roman character sets, line drawing character set, cartridge tape capability and five extra cartridges to be operated over phone lines:

2641A	APL Display Station
-001	Adds Lower Case Roman Character Set
-007	Adds Cartridge Tape Capability
-013	Adds Five Mini Cartridges
-202	Adds Line Drawing Character Set
13232N	Adds 103/202 Modem Cable—15 ft.

PRICE IN U.S.A.: 2641A, $4100. 2641A as above, $6115.

MANUFACTURING DIVISION: DATA TERMINALS DIVISION
19400 Homestead Road
Cupertino, California 95014 U.S.A.

A Dynamic Incremental Compiler for an Interpretive Language

by Eric J. Van Dyke

APL OFFERS THE USER a rich selection of primitive functions and function/operator composites. Powerful data structuring, selection, and arithmetic computation functions are provided, and their definitions are extended over vectors, matrices, and arrays of larger dimension, as well as scalars.

Evaluation of complex expressions built from such terse operations is necessarily quite involved. Code must be generated and executed to apply primitive functions to one another and to data atoms, with whatever type checks and representation conversions are required. Nested iteration loops must be created to extend the scalar functions over multidimensional array arguments, and these must include data conformity and index range checks.

All of this gathering and checking of information concerning data/function interaction and loop structure—and its high overhead expense—is, in the typical naive APL interpreter, simply thrown away after the execution of a statement. This is because the nature of APL is dynamic. Attributes of names may be arbitrarily changed by programmer or program. Size, shape, data type, even the simple meaning of a name (whether a data variable, shared variable, label, or function), are all subject to change (Fig. 1). Assumptions cannot be bound to names at any time and be counted on to remain valid on any subsequent loop iteration or function invocation. For this reason, APL has traditionally been considered too unstable to compile.

From this dilemma—high cost and wasted overhead that penalize interpretation but instability that prevents compilation—grew the dynamic incremental compiler of APL\3000.

Compile Only as Required

The APL\3000 *dynamic incremental compiler* is an interactive compiler/interpreter hybrid. It is a compiler that generates and saves executable object code from a tree representation of each new APL expression for which none already exists. (In general, each assignment statement, branch, or function invocation is considered an expression.) It is also an interpreter that immediately evaluates every expression of a statement or function. Whenever possible, previously compiled and saved code for an expression is re-executed. Only when absolutely necessary is new code generated. Thus stable expressions are compiled, while those with dynamically varying attributes and those that are executed only once are, in essence, interpreted. The overhead of new code generation is borne only when necessary, often only once. This scheme of infrequent overhead provides justification for costly optimizations, including the dragalong and beating discussed below, that lead to more efficient code.

A balance between compiling and interpretation is accomplished through the generation and execution of *signature code*, binding instructions that are emitted before the code for an expression. Their purpose is to specify and check the attributes that are bound into the following code, that is, constraints that may not change if the compiled code is to be re-executed. Signature instructions are generated that test index origin (0 or 1), meaning of names (whether data variable, shared variable, or otherwise), type and dimensions of expressions (representation, size, and shape), access information for data (origin and steps on each dimension), and run-time index bounds checks.

These signature instructions are bypassed on the first execution after compilation, when all assumptions are guaranteed satisfied. On subsequent executions, the signature code is used to test the validity of the code that follows. If these assumptions are found to be invalid, the code "breaks". Execution is returned to the compiler and code with a new set of assumptions is generated (Fig. 2). On recompilation, an expression is assumed unstable and a not-so-

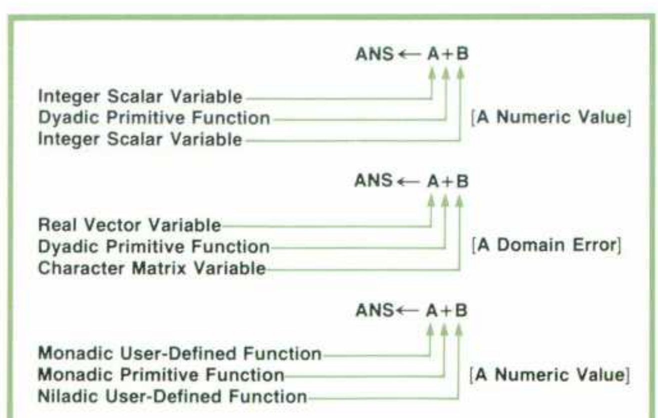


Fig. 1. *APL is dynamic. Attributes of names may be arbitrarily changed by the programmer or by a program. For this reason, APL has been considered impossible to compile.*

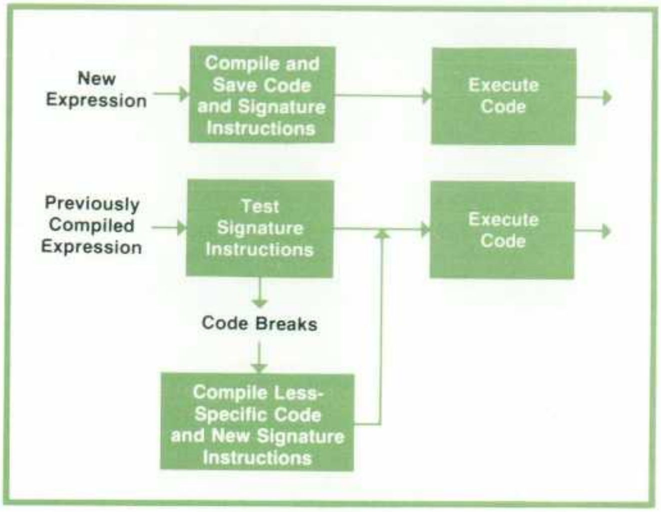


Fig. 2. *In APL\3000, expressions are compiled when first encountered. Along with the compiled code signature code is generated, specifying constraints that must be met if the code is to be re-executed. This signature code is tested on subsequent invocations of the expression, and if the constraints are not met, recompilation is required.*

specific but somewhat slower and less dense form of code is generated. Further changes may not force a recompilation.

Wait as Long as Possible; Do as Little as Necessary

The secret to compiling efficient code is in gathering, retaining, and exploiting as much information about the entire expression as possible before generating code. The more context that can be recognized, the more specific "smarts" can be tailored into the code. For this reason, the APL\3000 compiler operates in two distinct functional passes: context gathering and code generation.

The context gathering, or foliation, phase of compilation is a complete bottom-up traversal of the expression tree. Fig. 3 shows an example of such a tree. Description information is associated with each of the constant and variable data nodes—the leaves of the tree. These descriptions are then "floated" up to interact with the parent node. Descriptions are revised and attached to the corresponding node as necessary to suit the result. This process continues as descriptions are gathered and carried up through each function or operator node toward the root. Attached to the final assignment or branch node will be a context description for the entire expression. Fig. 4 shows the foliated tree for the expression of Fig. 3.

The information created by this foliation process consists of a set of auxiliary description nodes attached to each node in the expression tree. Each of these description groups contains the attributes of the result of the expression to which it is attached, as modified by that function and those below. First in the set of descriptor nodes is a single RRR node, which

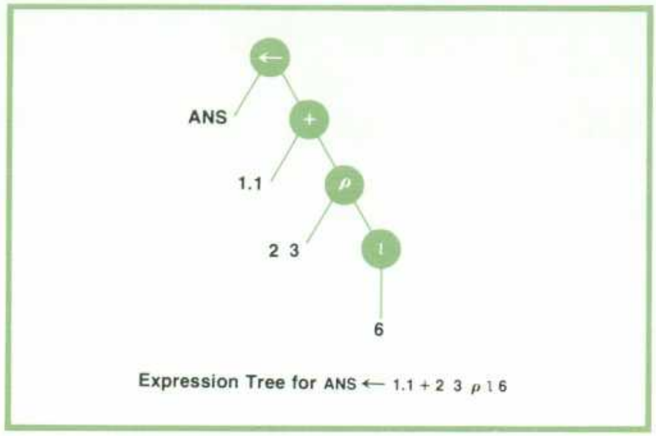


Fig. 3. *The tree representation of an expression. The APL\3000 compiler traverses this tree twice, once for context gathering and once for code generation.*

describes the general structure of the current expression: RANK (number of dimensions—for scalar, 0), REPRESENTATION (internal data type), and RHOs (size of each dimension—for scalar, there is none). Linked to the RRR node is a chain of DELOFF nodes, or data access descriptions, at least one for each non-scalar data item in the expression. A DELOFF node indicates the order in which an item is accessed and stored—row major, for example—by means of an OFFSET (origin), and a DEL (step) for each coordinate. Notice that these descriptions are independent of the data; storage need not be accessed during this foliation process.

Frequently, data storage is shared. In such cases, multiple descriptors are created, perhaps with differing access schemes. Each addresses the same shared area. A common form of vector data created by the INDEX GENERATOR function is the arithmetic progression vector (APV). This vector may be completely represented by its descriptor; no data area is necessary at all. For example, 2+3×ι4 requires only the descriptor:

RHO: 4 OFFSET: 5 DEL: 3

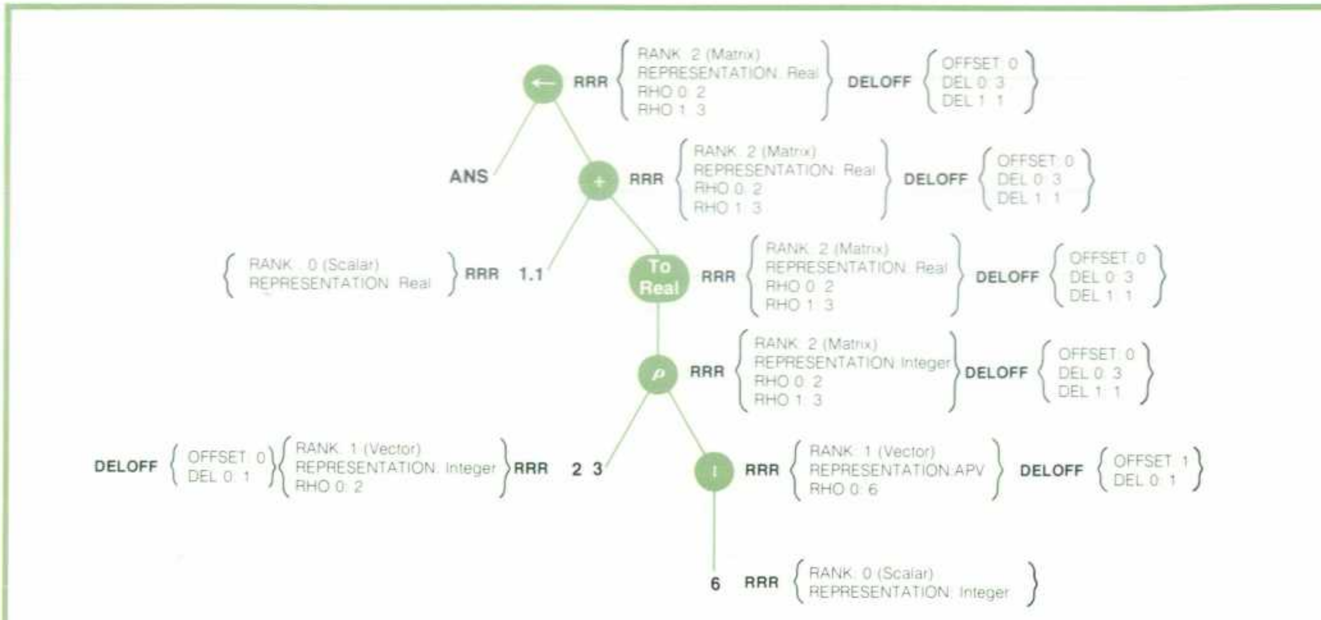
to represent the values 5 8 11 14.

Dragalong and Beating

It is the gathering and manipulation of these data-independent descriptors, following the dragalong and beating strategies developed by Abrams,[1] that makes possible the extensive optimizations incorporated in APL\3000.

Dragalong, the strategy of deferring actual evaluation as far as possible up the expression tree by gathering descriptions, avoids the naive interpreter's usual one-function-at-a-time "pinhole" evaluation. Instead, the code for a collection of parallel functions, including their associated loops, can be generated and executed simultaneously. Fig. 5 compares naive with dragged code.

Beating, the application of Abrams' subscript cal-



Foliated Expression Tree for ANS ← 1.1 + 2 3 ρ ι 6

Fig. 4. *Foliated expression tree results from the context gathering phase of compilation. Auxiliary description nodes contain the attributes of the sub-expression to which they are attached.*

culus to a deferred expression when evaluation is finally required, produces the desired results for certain APL functions by description manipulation alone. In such cases, the original data is shared with the beaten result, making it unnecessary to copy the data in a different form. Thus data is touched only when and only as much as necessary. (Data sharing is described in more detail in the article beginning on page 6.) SUBSCRIPTION, RESHAPE, RAVEL, TAKE, DROP, REVERSAL, and monadic and dyadic TRANSPOSE are the functions to which beating optimizations may be applied (see Fig. 6).

The dragalong and beating strategies can significantly reduce the amount of data access and storage, computation and looping overhead, and often temporary storage required in the evaluation of an expression.

An independent context gathering pass during compilation provides an opportunity for a number of specific optimizations in addition to dragalong and beating. For example, a pair of adjacent monadic RHO nodes can be recognized as a new internal RANK function. The result is merely the rank of the argument as indicated by its description, eliminating the need for an intermediate rho vector (see Fig. 7). Similarly, successive CATENATE nodes can often be incorporated into a new multi-argument POLYCAT function, eliminating the superfluous data moves and intermediate storage that would normally be required (Fig. 8).

Naive

```
INITIALIZE INDEX 1 AND LIMIT
WHILE INDEX 1 ≠ LIMIT DO
    BEGIN
    TEMPORARY [INDEX 1]←B[INDEX 1]×C[INDEX 1]
    INCREMENT INDEX 1
    END
INITIALIZE INDEX 2
WHILE INDEX 2 ≠ LIMIT DO
    BEGIN
    ANS [INDEX 2 ]←A[INDEX 2]+TEMPORARY[INDEX 2]
    INCREMENT INDEX 2
    END
```

Dragged

```
INITIALIZE INDEX AND LIMIT
WHILE INDEX ≠ LIMIT DO
    BEGIN
    ANS[INDEX]←A[INDEX]+B[INDEX]×C[INDEX]
    INCREMENT INDEX
    END
```

Fig. 5. *Evaluation of an expression is deferred as long as possible. This strategy, called dragalong, makes it possible to generate and execute the code for a number of parallel functions simultaneously, avoiding the naive interpreter's one-function-at-a-time evaluation. Shown here is a comparison of naive with dragged code for ANS←A+B×C. A, B, and C are conformable vectors.*

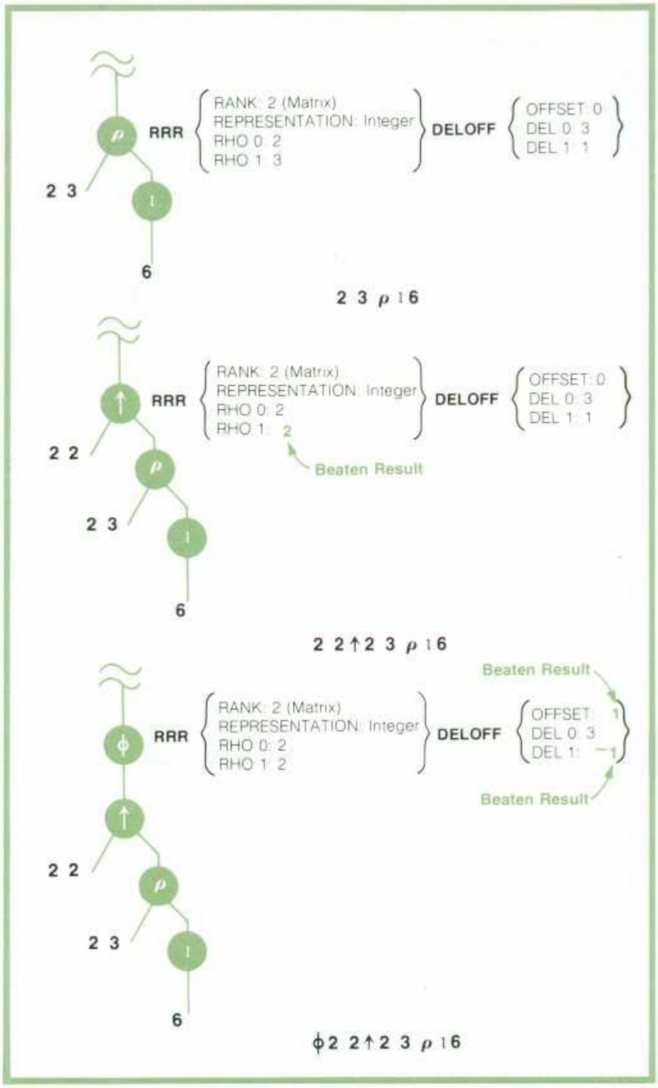


Fig. 6. *When evaluation is finally required,* beating, *or the application of the subscript calculus to a deferred expression, may produce results by description manipulation alone. Here TAKE (↑) and REVERSAL (ϕ) are applied to descriptions for a simple expression. The dragalong (see Fig. 5) and beating strategies can significantly reduce the computation and storage required in the evaluation of an expression.*

Code Generation

When the compiler is finally forced to materialize an expression—either the root has been reached, or the compiler can drag no farther for one reason or another—code is emitted. This code generation pass is a second independent walk of the foliated tree with dragged and beaten descriptions attached, this time from the top down, generating and saving executable code for the expression. By exploiting the context descriptions that have been gathered up the tree from each node, specifically tailored code can be generated. Because APL in general deals with arrays, this process also usually involves the construction of loops.

APL\3000's target machine is a software/firmware emulator implemented on the HP/3000. The instruction set, in addition to loads, stores, and loop and index controlling instructions, includes a set of high-level opcodes that match the APL primitive scalar functions. Code generation from an expression follows a recursive descent of the tree: an instruction to set up a storage area for the result (typically a temporary) is emitted, followed by a reverse Polish sequence of data loads and operations, and finally a store into the result, all nested within the necessary loops.

Any instruction that has the potential to fail carries within it a syllable number that provides the machine with a pointer to the original source in case of an error, allowing for recompilation on binding errors or message generation on user errors.

The descriptions at the root node completely describe all index variables and iteration loops to be generated. Each DELOFF node, with optimizations beaten in, describes the initialization (OFFSET) and stepping (DEL) of an index register. The loops, one for each dimension of the result, in general, are derived from the RRR in conjunction with a selected DELOFF. Loops are all of a basic structure:

```
INITIALIZE ALL INDEX REGISTERS
INITIALIZE LIMIT REGISTER
WHILE CHOSEN INDEX ≠ LIMIT DO
  BEGIN
  INITIALIZE LIMIT REGISTER
  WHILE CHOSEN INDEX ≠ LIMIT DO
    BEGIN
      ⋮
    (Indexed Expression Code)
      ⋮
    INCREMENT ALL INDEX REGISTERS
    END
  INCREMENT ALL INDEX REGISTERS
  END
```

Equality, unlike > and <, is a consistent termination condition for loops that may run in any direction. For each loop, a DELOFF node is selected to serve as the loop-controlling induction variable. Because of their special uses, certain indexes are not eligible (those for

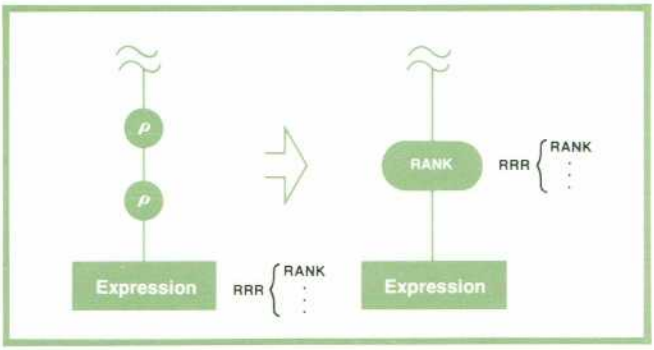


Fig. 7. *The context gathering pass provides an opportunity for specific optimizations, such as recognizing a pair of adjacent monadic RHO nodes as the new internal RANK function.*

A Controller for the Dynamic Compiler

by Kenneth A. Van Bree

The controller for the dynamic compiler performs all of the tasks an interpreter for APL must perform, such as handling user input and editing, sequencing between lines of a function, calling and returning from user-defined functions, and handling errors. In addition, the controller handles the generation and re-execution of compiled code for APL statements.

One of the guiding assumptions in the design of the controller was that code for a particular statement could be compiled once and would remain valid for many re-executions of that statement. This assumption was based on the observation that most APL programmers do not take full advantage of the dynamic capabilities of APL. Changes in the value or size (number of elements) of a variable are frequent, but changes in the shape or representation of a variable are rare. For this reason, the controller has been designed to re-execute compiled code as quickly as possible, while still maintaining the flexibility needed to perform all the other duties related to controlling an interactive language such as APL.

The controller consists of five interacting modules as shown in the diagram. Each module performs a subset of the duties related to controlling the compiler, and any module can call on any other module to perform a task that it cannot do itself. The normal flow of control for an APL expression input by the user (in calculator mode) is as follows:

Text for the expression is input by the user through the user input and editing module. This module is in charge of all interactions with the user, and before control leaves this module, all text that the user enters is converted into an internal form called S-code. S-code is a compact form of the text, with each identifier replaced by an internal short name for easy reference. The actual text that the user enters is not saved, but is regenerated from S-code if needed.

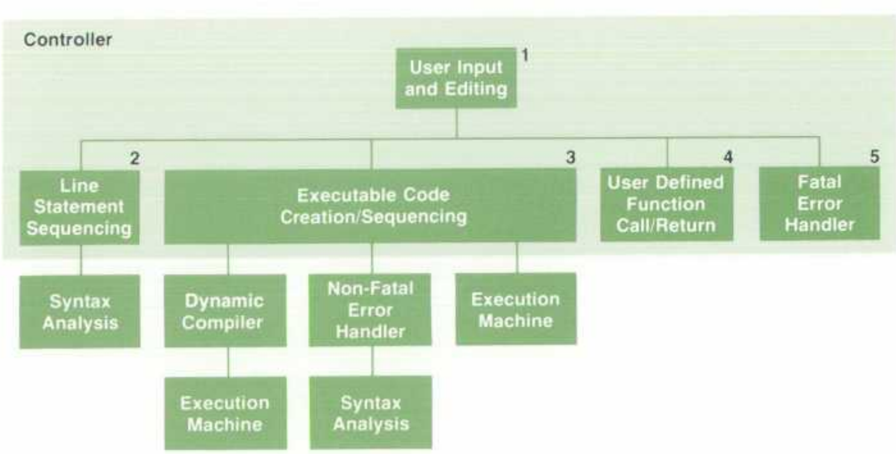
Once S-code has been created, control is passed to the line statement sequencing module, which handles the dynamic flow of control between lines and statements in APL. As each statement is executed, this module checks to see whether it has been executed before. If a statement has never been executed before, a syntax analysis is done on the S-code for that statement. The result of the syntax analysis is one or more syntax trees called D-trees. Each D-tree represents the largest part of an APL statement that can be guaranteed to have no side effects. For example, in the statement A←B+C, if C is a user-defined function, then the statement will be broken up into two trees. The first tree will materialize the function C into a temporary variable, and the second tree will add the results of C to B and assign the sum to A.

As soon as D-trees have been created for a statement, control is passed to the executable code creation/sequencing module. Within this module, each D-tree for a statement is examined in sequence, and if it does not represent a function call, it is passed to the dynamic compiler. The compiler turns each D-tree into a block of executable code called E-code. The compiler calls the execution machine directly to execute the E-code that it has created.

Once a valid block of E-code has been created from a D-tree the executable code creation/sequencing module is in charge of storing that E-code block for later reference. As each D-tree is compiled, the E-code block created is used to replace the D-tree. When all trees for a statement are compiled there will exist a series of E-code blocks that represent the statement. On subsequent executions of a statement, the E-code blocks are retrieved and given directly to the execution machine. If the code contains a non-fatal error such as a change in representation or rank of a variable, the execution machine returns a non-fatal error indication to the executable code creation/sequencing module, which calls the non-fatal error handler to correct the problem. The non-fatal error handler recreates a D-tree for the part of the statement affected by the non-fatal error. New E-code is then compiled with the non-fatal error corrected, and the new E-code block is saved in place of the one in which the error was found.

If the executable code creation/sequencing module detects that a particular D-tree represents a function call, then control is passed to the user-defined function call and return module. If the line statement sequencing module detects a function return, it can also pass control directly to the user-defined function call and return module.

If any of the other modules detects a fatal error, such as an undefined variable or a syntax error, control is passed directly to the fatal error handler. This module suspends execution, prints an error message for the user, and then returns control to the user input and editing module to wait for input from the user.



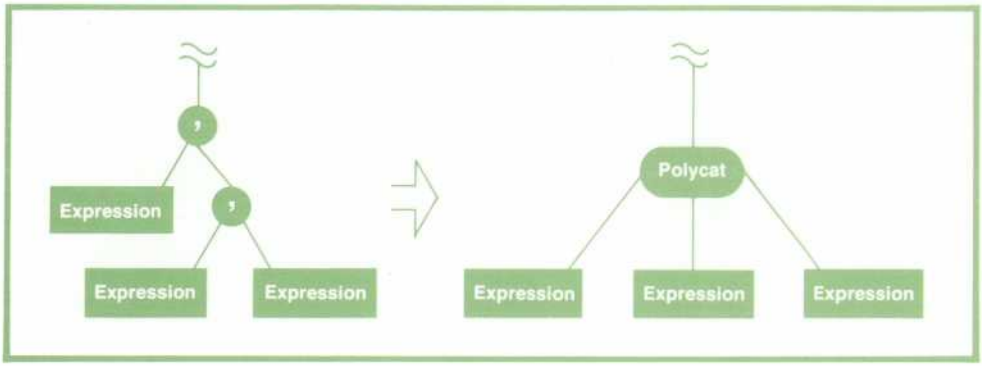


Fig. 8. *Another optimization that can be effected during context gathering is combining successive* CATENATE *nodes into a new internal* POLYCAT *function.*

single-element arrays that will never be incremented, for example, or the left indexes of COMPRESS and EXPAND, which are incremented asynchronously).

A limit for each loop, calculated as OFFSET + RHO × DEL (on the appropriate coordinate, from the chosen induction variable) plus the current induction variable, is also created in a register. Except for the outermost (or only) loop limit, which may be constant, the limit value must be calculated at execution time. Initialization values and increments for all indexes correspond to the OFFSETs and DELs of their associated DELOFF descriptors. Fig. 9 shows the code generated for a vector expression.

A number of optimizations are performed prior to the generation of loops. Except for actual display, an expression represented as an arithmetic progression vector (APV) requires no evaluation loop at all; its description completely specifies the result. Redundant index variables, which would run in parallel, are shared by collecting those DELOFF nodes having identical attributes into a single register. If, according to the descriptors, a loop is unnecessary, as is often the case with row-major compact storage, it is collapsed, subsumed by the next outer loop.

In addition, certain improvements in the code can be made. Unlike larger data structures, in which data can be partially destroyed if an error is encountered, scalar and single-element expressions can be generated without assignment to an intermediate temporary variable, eliminating the setup, some use of storage area, and the resulting data swap. Occasionally, when the result produced from such a unit expression involves itself, a new data area need not be set up at all. Instead, the old name is retained for the result of the expression. Subexpressions yielding a scalar or single-element array within the scope of a loop can frequently be materialized, or assigned into a temporary cell, outside the loop, eliminating their repeated evaluation. The more complex argument to an OUTER PRODUCT operator can similarly be constrained to an outer code loop, affording it less frequent evaluation.

Hard and Soft Code

The code generated by APL\3000 is of two types. Initially, hard or tight code is produced. In this style of code, the RHOs, OFFSETs, and DELs, as well as RANK and REPRESENTATION are bound into the instructions as constants. If this specific form of code has broken and a recompilation is required, more general soft or loose code is generated, in which only the RANK and REPRESENTATION are bound. RHOs, DELs, and OFFSETs may be calculated in registers at run time. Thus the dimensional attributes of an array may dynamically change without invalidating the code again.

```
SET UP STORAGE AREA FOR RESULT TEMP
INITIALIZE STORING INDEX TO 0
      (OFFSET FOR ANS AND TEMP)
INITIALIZE VECTOR ACCESSING INDEX TO 2
      (OFFSET FOR VECTOR BEATEN BY ⌽)
INITIALIZE APV ACCESSING INDEX TO 1
      (OFFSET FOR ι3)
INITIALIZE LIMIT TO 3
      (RHO × DEL + OFFSET + STORING INDEX)

WHILE STORING INDEX ≠ LIMIT DO
      BEGIN
      LOAD APV ACCESSING INDEX
      INTEGER LOAD OF VECTOR [VECTOR ACCESSING INDEX]
      INTEGER MULTIPLY
      CONVERT TO REAL
      REAL LOAD OF CONSTANT 1.1
      REAL ADD
      REAL STORE INTO TEMP [STORING INDEX]
      INCREMENT STORING INDEX BY 1
          (DEL FOR ANS AND TEMP)
      INCREMENT VECTOR ACCESSING INDEX BY ¯1
          (DEL FOR VECTOR BEATEN BY ⌽)
      INCREMENT APV ACCESSING INDEX BY 1
          (DEL FOR ι3)
      END
SWAP TEMP INTO ANS
```

Fig. 9. *When the compiler can drag no farther it emits code. The code generation phase is a second traversal of the (now foliated) expression tree. Because APL in general deals with arrays, code generation usually involves the construction of loops. Shown here is the code generated for the expression* ANS←1.1 + (⌽VECTOR)×ι3. VECTOR *is an integer vector of length 3.*

Hard

```
SET UP STORAGE AREA FOR RESULT TEMP
INITIALIZE STORING INDEX TO 0
INITIALIZE VECTOR ACCESSING INDEX TO 2
INITIALIZE LIMIT TO 3
WHILE STORING INDEX ≠ LIMIT DO
      BEGIN
      LOAD 1
      LOAD VECTOR[VECTOR ACCESSING INDEX]
      ADD
      STORE INTO TEMP[STORING INDEX]
      INCREMENT STORING INDEX BY 1
      INCREMENT VECTOR ACCESSING INDEX BY ¯1
      END
SWAP TEMP INTO ANS
```

Soft

```
SET UP STORAGE AREA FOR RESULT TEMP
INITIALIZE STORING INDEX TO 0
INITIALIZE VECTOR ACCESSING INDEX TO
      (RHO−1) × DEL + OFFSET FROM VECTOR
INITIALIZE VECTOR ACCESSING INCREMENT TO
      ¯DEL FROM VECTOR
INITIALIZE LIMIT TO RHO FROM VECTOR
WHILE STORING INDEX ≠ LIMIT DO
      BEGIN
      LOAD 1
      LOAD VECTOR [VECTOR ACCESSING INDEX]
      ADD
      STORE INTO TEMP[STORING INDEX]
      INCREMENT STORING INDEX BY 1
      INCREMENT VECTOR ACCESSING INDEX BY
          VECTOR ACCESSING INCREMENT
      END
SWAP TEMP INTO ANS
```

Fig. 10. *Code generated is of two types. Initially, hard code is produced. If this code later breaks, more general soft code is generated. Shown here is hard versus soft code for the expression ANS←(⌽VECTOR) + 1. VECTOR is an integer vector of length 3.*

For this more flexible form of instruction a price is paid in terms of speed and code bulk, but this overhead cost rarely approaches that of an entire recompilation every time a RHO, OFFSET, or DEL changes. Notice that RANK and REPRESENTATION must always be bound hard. RANK, which specifies the maximum number of loops to be generated, must have a constant value at compile time. REPRESENTATION must be known to determine the data type of the instructions issued. A change in either of these attributes always forces a new compilation.

Fig. 10 compares hard and soft code emitted for a vector expression.

Reference

1. P.S. Abrams, "An APL Machine," PhD dissertation, SLAC Report No. 114, Stanford University, February 1970.

Eric J. Van Dyke

Eric Van Dyke began writing compilers right after he received his BA degree in information sciences from the University of California at Santa Cruz in 1974. After joining HP in 1975, he helped implement the dynamic incremental compiler for APL\3000. Eric is a California native, born in Palo Alto, and now lives in Los Altos. He's single, and has a passion for wilderness mountaineering, including climbing, hiking, skiing, and leading Sierra Club trail maintenance and clean-up trips. He's also a student of American folk music and jazz and folk and modern dance.

Extended Control Functions for Interactive Debugging

by Kenneth A. Van Bree

Several system functions facilitate debugging and program development in APL. Using the function ⎕SS (set stop) it is possible to stop on any or each line of a function or on return from the function. The ⎕ST (set trace) function allows the last result calculated on a line to be displayed along with the function name and line number. This is helpful for observing program flow. The ⎕SM (set monitor) function allows the user to monitor the number of times that a function and/or line has been executed, along with the amount of CPU time spent in each line, and the total CPU time spent in the function. These functions can be

(continued on page 24)

used to determine where the majority of the CPU time is being spent on a particular problem and which lines of a program have never been executed. All of the monitoring facilities can be turned on or off and queried under program control.

One reason that program development is so easy in APL is that the entire power of APL is available to the user during program debugging. When the APL system detects an error in a user program (for example, an attempt to read a variable that hasn't been given a value), the program is halted and an error message is written on the user terminal. The error message tells the user the type of error (a VALUE ERROR in this example) along with a pointer to where the error was detected. The APL system then returns control to the terminal so the user can try to correct the error. At this point the state indicator (SI) may be displayed. The state indicator is a pushdown list (i.e., stack) of all the user-defined functions that have been called but have not yet completed execution. The state indicator displays not only the names of the functions that have been called, but also the line number on which execution was suspended. In addition, a list of all the local variables can be obtained for each function that has been called but not completed. The function in which the error was found is the topmost entry on the SI and is called a suspended function. Other functions on the SI are called pendant functions.

While computation is suspended, the user has the full power of APL available to him for debugging. The suspended function (or any other function that is not pendant) may be edited, and any variable that is available within the suspended function may be interrogated or redefined. A new computation may be started by calling another function, or in most cases the suspended computation may be resumed from the line at which it was suspended or any other line. If for some reason the user does not wish to fix the error, the SI can be cleared, or the entire workspace including the SI can be saved for later reference.

The flexibility and power available to the user during debugging make it possible to detect and correct multiple errors during the course of the computation. This means that programs often run to completion the first time they are called, because most errors can be fixed as they are detected. A recent study of APL in Europe[1] showed that the conciseness of APL coupled with its ease of debugging produced a 3:1 improvement in programmer productivity over such languages as PL/I and COBOL.

Extended Control Functions

The state of an APL computation can be displayed at any time by interrupting the computation (by sending the ATTENTION character) and displaying the state indicator through the use of the commands)SI or)SIV. The state indicator shows all of the functions that have been called but have not yet completed execution, along with the variables that are local to those functions. The current environment consists of the variables that can be accessed within the topmost function on the stack, along with the chain of control represented by the function calls that appear on the SI. Normally, within APL, any computation must be done in the current environment. For example, if the function F (which has local variable V) calls function G (which also has local variable V), and computation is suspended within G, the SI might appear as follows:

```
      )SIV
G[3] *    V
F[2]      V
```

In this environment the value of variable V is whatever has been assigned within function G. The value of V within function F has been shadowed (by the local variable V within G) and is not accessible within the current function. All names accessible from function G make up the environment of G, and the local variable V of function F is not in the environment of G. Furthermore, it is not possible to resume execution of function F without first completing function G, since the SI operates strictly on a last-in-first-out basis.

Through the use of the extended control functions of APL\3000 it is possible to access variables and resume execution in environments other than the current environment. The concept of multiple environments is not new,[2] but it has never been implemented in APL before. APL\3000 allows up to 16 environments to be available at one time. Each environment has its own state indicator, and control can be passed from one environment to another through the use of the extended execute (⍎) function. Although the normal SI in APL obeys a strict stack discipline, the environments of APL\3000 may create one or more computation trees. This allows the creation of environments that share a portion of their SI. When this happens, it is no longer possible to maintain a stack discipline for the SI, and a set of pointers must be maintained that links each function call to its calling function. The extended control functions maintain a stack discipline for the SI unless the user explicitly calls for a tree-like control structure. The overhead paid for the extended control capability is minimal unless it is invoked by the user. In the above example, the environment within function F can be captured by using the system function ⎕CSE (capture stack environment).

```
      ⎕CSE 2    ⍝ Capture second function name on SI
1               ⍝ The environment number is 1
      )SIV 1    ⍝ Display the SI for environment 1
F[2]       V
```

Environment 1 now shares a part of its SI (namely the function F and its local variable V) with the current environment displayed earlier. Any arbitrary expression can be evaluated in the environment of function F through the use of the extended execute function. For example, the variable V within function F may be assigned the value 3 as follows:

```
1⍎'V←3'
```

Evaluating an expression in environment 1 (or any other environment) is equivalent to evaluating the expression in calculator mode with execution suspended in that environment. Execution can be resumed within function F by evaluating an expression that results in a branch. For example:

```
1⍎'→2'
```

The extended control functions in APL\3000 can be used for purposes other than debugging. Since environments can be captured (using ⎕CSE) and released (using ⎕RSE) under program control, it is possible to implement such advanced programming concepts as backtracking, co-routines, and so on, which have been difficult or impossible to implement in APL before.

References

1. Y. LeBorgne, "APL Usage in Europe: Scope and Value," Proceedings of APL 76, Ottawa, Canada, September 1976, pp. 259-266.

2. D.G. Bobrow and B. Wegbreit, "A Model and Stack Implementation of Multiple Environments," Communications of the ACM, Vol. 16, No. 10, October 1973, pp. 591-603.

CRT Terminal Provides both APL and ASCII Operation

by Warren W. Leong

MODEL 2641A APL DISPLAY STATION (Fig. 1) is a special CRT terminal designed to serve as the principal user interface for APL\3000. APL operation plus extensive data communications capabilities allow the terminal to be used with APL interpreters/compilers that exist on a variety of computer systems, especially the HP 3000. ASCII operation is provided to retain compatibility with HP 2640-Series CRT Terminals.

The 2641A provides a superset of the functions available with the 2645A Display Station. These include dual cartridge tape units, extended editing features, extended data communications, modular firmware implementation, and eight user-defined soft keys. A new, faster microprocessor provides the control for the standard as well as the extended features.

APL Features

Major features of the 2641A APL Display Station are: display of the APL character set, display of the APL overstrike character set*, display of APL underlined characters, and non-destructive spaceover. These features are accessible during the terminal's APL mode.

The high-resolution display of 2640 Series Terminals[1,2] provides a clear and easily readable rendition of the standard APL characters as well as the more intricate overstrike characters (Fig. 2). There are two separate APL character sets: a 128-character APL graphics set and a 64-character APL overstrike

*Many APL primitive functions are called by striking one APL symbol, then backspacing and overstriking the first symbol with a second symbol. The combination forms a new APL symbol. The APL overstrike character set makes it possible for the 2641A to display such combinations of basic APL symbols.

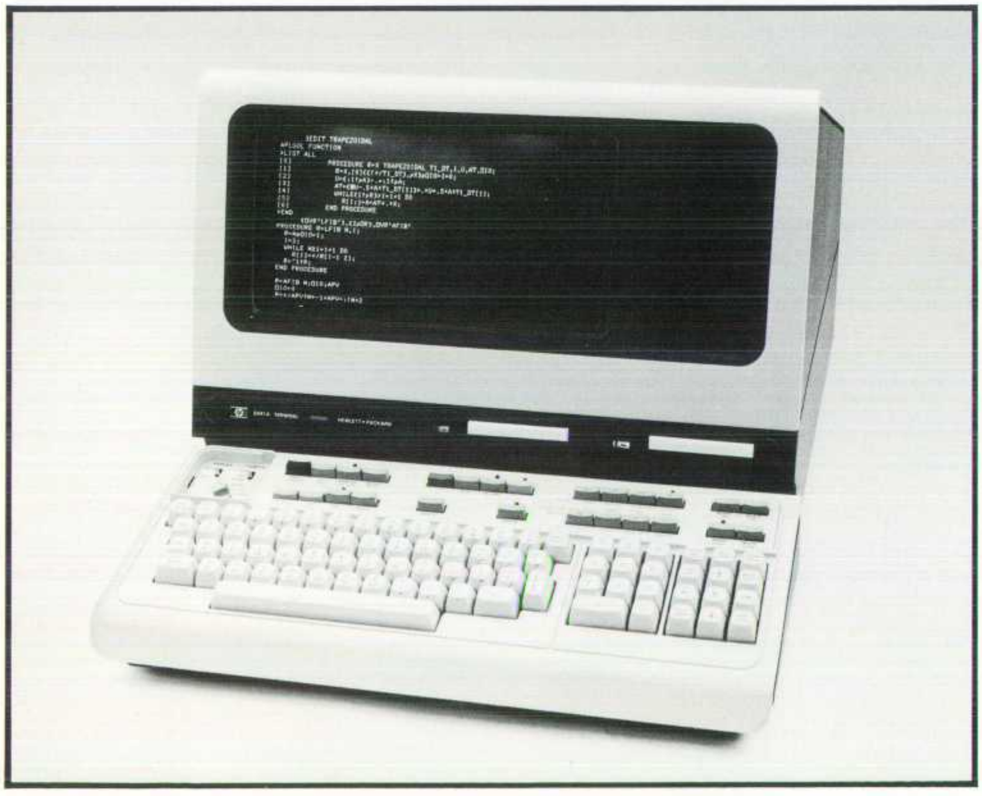


Fig. 1. *Model 2641A APL Display Station is designed to serve as the principal user interface for APL\3000 and other APL systems. It has both APL and ASCII modes of operation.*

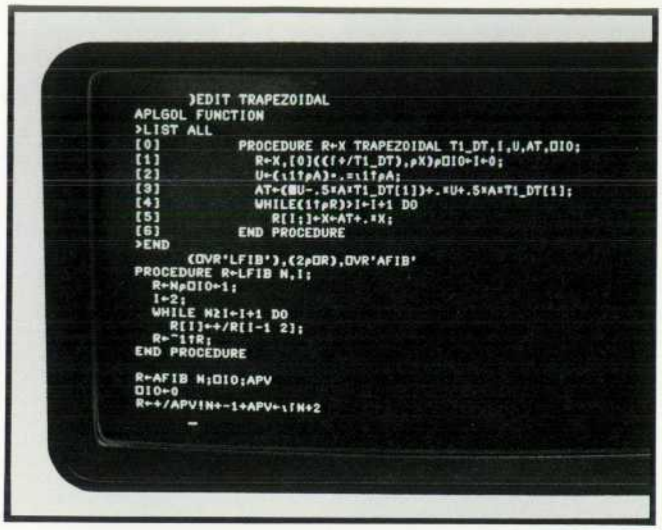


Fig. 2. *Standard 2641A character sets are the 128-character APL set, a 64-character APL overstrike set, and a 64-character upper-case Roman set. An optional fourth character set may be a mathematical symbol set, a line drawing set, a large character set, or a user-designed set.*

graphics set (Fig. 3). Each set is programmed into bipolar ROMs. The APL graphics set follows commonly accepted industry standard code assignments. The APL overstrike graphics set is used internally by the terminal to display the overstrike characters and its code assignment is dependent on terminal requirements. As each valid overstrike keystroke sequence is completed the proper overstrike character is displayed on the screen. However, the actual overstrike character sequence is transmitted to the computer when in character mode or is stored in the display memory for later transmission when in block mode.

The 2640 Series Terminals can support up to four independent character sets. Since the 2641A APL Terminal includes as standard the APL set, the APL overstrike set, and the ASCII set, it has room for one additional character set. Currently this additional set can be a mathematical symbol set, a line drawing set, a large character set, or a set of the customer's own design.

The keycaps have APL legends on their top faces and ASCII legends, when they differ, on the front faces (Fig. 4). This allows unambiguous operation whether operating in APL or ASCII mode. The keyboard code assignment is bit pairing*, rather than typewriter pairing*, to retain compatibility with the 2640B and 2645A Terminals. The shift 0 (zero) position is re-assigned to mean ∧ in APL and _ in ASCII; this provides full APL compatibility for users when switching between bit and typewriter pairing layouts.

Firmware

The controlling feature of the 2641A APL Display Station is the firmware, or microprograms stored in ROM. All of the characteristics of the terminal are defined by microprogramming the internal microprocessor. These characteristics include switch selection or computer selection via escape sequence of the two operating modes, APL or ASCII, overstrikes that are recognized by the terminal, block transfers of APL program and data statements, and editing features during APL mode.

The first consideration was how to integrate the APL operational requirements into the base product, the 2645A. Since many of the features of APL were distinctly different from normal operation, it made sense to define an APL mode for APL operations. In APL mode the APL character set is normally displayed instead of the ASCII character set. Any attempt to overstrike an APL character results in the display of a character from the overstrike set. Underlining of APL characters is done by means of shift F. Block transfers (via the ENTER key) take into account the overstrike character set and decompose these into APL characters separated by a backspace control code.

APL systems recognize several overstrikes. With

*Bit pairing: shift codes differ from unshift codes by one bit.
Typewriter pairing: codes follow an industry standard for certain typewriter terminals.

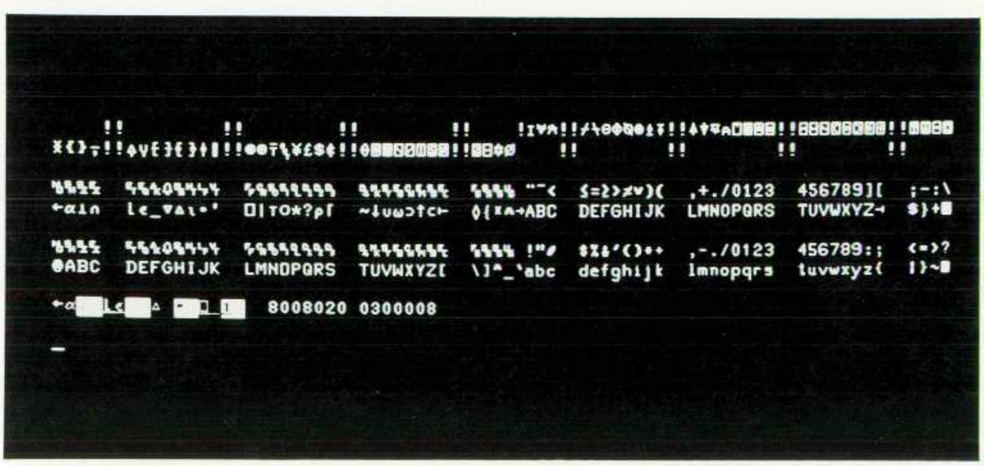


Fig. 3. *Standard 2641A character sets.*

Fig. 4. *2641A keys have APL legends on top and ASCII legends, when they differ, on the front faces.*

the 2641A, these overstrikes can be done at any time or in any order. Overstriking poses several complications for a raster-scan CRT terminal that dynamically allocates its memory and uses separate graphics sets for the normal and overstrike characters. An APL user may type several characters, then backspace to the beginning of the line and overstrike the required characters, or the user may complete each overstrike before proceeding to the next character. Backspacing, using the backspace key, does not delete characters previously entered and forward spacing using the space bar does not erase characters that are being spaced over.

The basic algorithm for overstrikes directs the terminal to monitor each byte that it writes to the display. In APL mode, the terminal checks the current and new characters being typed in the same display position and determines whether the new character just overwrites the old (only when the old character is a blank), whether the old character is replaced by a new character from the overstrike set, or whether the old character remains unchanged (the new character is a blank). Overstrikes are allowed only in APL character fields. If the cursor is in a non-APL field, such as Roman, then the terminal performs ASCII operations rather than APL operations, although the operating mode is APL.

When the old and new characters form a valid overstrike such as ' and ., then the composite ! is displayed. If an invalid pair is overstruck, then an OUT character is displayed, providing a clear indication that an error has been made.

The underline overstrike (shift F) for APL is normally restricted by APL systems to the alphabetic characters and a few of the special characters. The 2641A can underline any APL character. The underline overstrikes are not a part of the character ROMs. Instead, the underline feature of the terminal's display enhancement section is used to simulate the underline overstrike.

The underlining process begins when an APL character is displayed and the cursor is repositioned to the character. When the underline character (shift F) is typed, the firmware provides the proper enhancements to underline the character.

Data Transfer

All display information, overstrikes, and underlines can be stored on the cartridge tape units, printed on a printer, or block transmitted to a computer system.

Block transfers during APL mode, from the display or the tape units, take into account the overstrike set and underline enhancements. In the case of overstrikes, the code from the overstrike ROM is used as an index into a look-up table for the two components of the overstrike. These two components are then transmitted with a backspace separating them. The underlined characters are transmitted with the proper codes: the character, then backspace, then underline. The OUT character is treated as a special case and causes five characters to be output: 0 backspace U backspace T.

Two types of printers are available for APL: bit pairing or typewriter pairing. Distinguishing the two are the code assignments of 19 of the characters. The 2641A allows the user to select either translation when directing APL data to a printer.

User-Defined Soft Keys

The 2641A has eight special-function user-definable soft keys, f1 through f8. These keys hold up to 80 ASCII characters that are specified by the user. This specification may be done interactively, with the old contents displayed while updates are done. The specification may also be done by escape sequence from a computer system or from the optional cartridge tape units.

After logging onto an HP/3000 Computer System having an APL\3000 subsystem, the user specifies the terminal type to be a 2641A by means of the)TERM HP command, and the system downloads the soft keys with the following commands:

Command:	)RESUME	)SI	)FNS	)VARS
Key:	f1	f2	f3	f4
Command:	ATTN	)EDIT	)LOAD	)SAVE
Key:	f5	f6	f7	f8

Now the user can invoke frequently typed system calls with a single keystroke. For instance, to edit a function named APL1, the user can press f6 to call the system editor, then type APL1, followed by RETURN, and be ready to edit. The user may also redefine these soft keys very simply.

Key f5 contains the ATTN command, which is useful during line editing. Suppose the user has typed a line of data but notices a mistake. To correct the error, the user first backspaces the cursor to the incorrect character:

A B C F E

Using the 2641A and APL\3000, the user then hits ATTN, which causes the APL terminal driver to send an escape sequence to clear the rest of the line:

A B C _

The user continues typing from this point to complete the data statement:

A B C D E _

The traditional method of editing is to position the cursor under the incorrect character, then send a line-feed to the computer and type the correct characters, producing a display like:

A B C F E
      D E

Note that the display can be confusing to read if several corrections have to be made in this manner. However, both methods of correction are allowed by the subsystem and the 2641A.

Extended Features

Editing features have been expanded to include character wraparound when the terminal is doing character delete or insert operations. Left and right margins may also be set. Extended I/O operations with the cartridge tape option include write, back-space, read, data comparisons, and data logging.

The data communications facility allows data rates up to 9600 baud, and multipoint capabilities that allow up to 32 terminals to share a single communications line. Self-test has been expanded to allow testing of the optional cartridge tapes and associated electronics as well as the multipoint communications option, cabling, and terminating instrumentation. Multipoint communications can even be tested up to the remote modem from the terminal keyboard.

Acknowledgments

This product relied on the flexible base provided by the designers of the 2645A Display Station: Tom Waitman, Ed Tang, Rick Palm, Greg Garland, Gary Staas and Bill Woo. Dave Goodreau, Jim Elliott, and Hans Jeans provided additional product definition assistance.

References

1. J.A. Doub, "Cost-Effective, Reliable CRT Terminal Is First of a Family," Hewlett-Packard Journal, June 1975.
2. R.G. Nordman, R.L. Smith, and L.A. Witkin, "New CRT Terminal Has Magnetic Tape Storage for Expanded Capability," Hewlett-Packard Journal, May 1976.

Warren W. Leong

Warren Leong has been involved with the firmware and character set design for the 2640B/C/N/S and 2645R/S CRT Terminals and the 2641A APL Display Station. He's been with HP since 1975. Born in San Francisco, Warren attended the City College of San Francisco and the University of California at Berkeley, graduating from the latter in 1975 with a BS degree in electrical engineering and computer science. He's single, a tennis player, and lives in Sunnyvale, California.

Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, California 94304

HEWLETT-PACKARD JOURNAL

JULY 1977 Volume 28 • Number 11

Technical information from the Laboratories of Hewlett-Packard Company

Hewlett-Packard Central Mailing Department
Van Heuven Goedhartlaan 121
Amstelveen-1134 The Netherlands
Yokogawa-Hewlett-Packard Ltd., Shibuya-Ku
Tokyo 151 Japan

Editorial Director • Howard L. Roberts
Managing Editor • Richard P. Dolan
Art Director, Photographer • Arvid A. Danielson
Illustrator • Susan E. Wright
Administrative Services, Typography • Anne S. LoPresti
European Production Manager • Dick Leeksma

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

MR C A BLACKBURN
JOHN HOPKINS UNIV
APPLIED PHYSICS LAB
JOHNS HOPKINS RD
LAUREL MD 20810

208107JOHNAAABROWAAA 181