Try APL2

for free

Version 2.00

Agreement

You may not enhance or modify this offering except for providing APL2 functions, variables, and operators to be used for promotion, demonstration, or education. TryAPL2 may not be used for commercial computing. You may not package TryAPL2 with another software product sold commercially. You may not use TryAPL2 with another software product sold commercially and designed for its use. You may not modify TryAPL2 to overcome restrictions. You may include TryAPL2 in a publication or book available commercially so long as the price of the publication or book does not include a fee for TryAPL2 beyond the cost of media and duplication. You may copy and distribute TryAPL2 files in their entirety. IBM's rights extend to all copies so made. Disk copies should be labeled with the information on the sample disk labels in the back of this document. You may charge a fee only to cover the cost of duplication, media, and distribution.

This offering is not covered by any warranty expressed or implied.

Contents

TryAPL2

Objective of the package

TryAPL2 is a vehicle for demonstrating and teaching the APL2 language. The package presents examples of APL2 solutions to problems in data processing, system design, mathematical programming, artificial intelligence, business, finance, and education. For the mathematics classroom, TryAPL2 provides an interactive learning environment for developing problem solving skills by exploring and discovering mathematical principles and algorithms.

The software included in this package is not a stand alone demo program. Rather, it is designed to be used as an introduction to APL2 programming and a vehicle to provide computational support for the teaching of other subjects. The package supports the full APL2 language, but it does not support the full APL2 environment. The differences between TryAPL and APL2 are listed in Appendix G.

What do you need to run this package?

TryAPL2 is designed to run on any IBM personal computer or Personal System/2 with the DOS operating system. Running on an IBM 4860 PCjr requires connecting a 4.7K resistor between Vcc (pin 40) and the test pin (23) of the 8088. You must have at least 512K of memory and a CGA, EGA, or VGA display for support of the APL2 character set. More memory may be required if you run resident programs. In particular, a monochrome screen is not supported. An IBM Personal Computer Math Coprocessor (80x87) is not required but is used if installed. TryAPL2 may not run on some PC clones and compatibles especially if the 80x87 is being emulated.

What files are supplied with TryAPL2?

TryAPL2 is distributed with the following files:

- READ.ME - installation instructions
- CONTRACT - agreement as shown in front of this document
- TRYAPL2A.EXE - self-extracting PKZIP file for the system
- TRYAPL2B.EXE - self-extracting PKZIP file for the workspaces
- TRYAPL2C.EXE - self-extracting PKZIP file for online documentation
- TRYAPL2D.EXE - self-extracting PKZIP file for printable documentation

TRYAPL2A when executed produces the following files which will fit on a 360K disk:

- CONTRACT.A - agreement as shown in front of this document
- APL2EGA.CPI - EGA VGA codepage information file
- APL2LCD.CPI - IBM PC Convertible codepage information file
- APL2FONT.COM - program for APL2 characters on the screen
- TRYA.EXE - the executable TryAPL2 program
- TRYAPL2.BAT - a batch file to run TryAPL2
- TRYAPL2N.BAT - a batch file to run TryAPL2 with alternate font
- TRYDOC.TRY - online documentation programs

TRYAPL2B when executed produces the following files which will fit on a 360K disk:

- CALENDAR.TRY - date computation programs
- DISPLAY.TRY - workspace for showing array structure
- GRAPHS.TRY - graphics examples
- IDIOMS.TRY - an APL2 idiom search program
- IE.TRY - expert system inferencing engine
- KEYS.TRY - keyboard set and display workspace
- OR.TRY - operations research network functions
- PRINT.TRY - printer support functions
- SEARCH.TRY - AI game playing programs
- STATS.TRY - statistics functions

TRYAPL2C when executed produces the following files which will fit on a 360K disk:

- TRYDOC.211 - online documentation file
- APL2VIO.DCP - APL2 characters for DOS full screen in OS/2

TRYAPL2D when executed produces the following files:

- TRYAPL2D.HPC - LaserJet version of documentation
- TRYTEMPL.HPC - LaserJet version of keyboard templates

PKZIP is a trademark of PKWARE, Inc., 7545 N. Port Washington Rd., Glendale, WI 53217, and is used with their permission under the IBM/PKWARE Corporate Contract EQD 340.

Performance of the package

TryAPL2 simulates the environment of the complete APL2/PC product. Therefore, the performance of the purchased product can be significantly better than the performance of TryAPL2. This is, in particular, true for system commands like `)LOAD` and `)SAVE` and for the program editor. Use of an IBM Personal Computer Math Coprocessor (80x87) is recommended for best performance.

What is APL2?

APL2 is different from most other languages in that it uses symbols to denote its operations rather than words. This, together with the fundamental use of array data, makes APL2 a powerful, concise, dynamic tool for problem solving.

APL has these distinguishing features:

- APL2 has a few simple rules so you can learn to write correct expressions immediately.
- APL2 deals with whole collections of data (arrays) at once.
- APL2 has a rich set of functions that can apply to whole arrays at one time without writing a "program."
- APL2 has operators that can modify functions, creating whole families of related functions in a uniform manner.

Everything you learn by using TryAPL2 can be used with any IBM APL2 implementation. Thus, when you learn how to use TryAPL2, you're learning much of what you need to make effective use of APL2/PC on the PS/2 as well as APL2 on the IBM S/370 S/390 and the IBM RISC System 6000.

The APL2 character set

APL2 is easiest to use when the APL2 symbols are inscribed on the keytops of your computer. In case they are not, templates for the keyboards that you can cut out and attach to your keyboard are supplied in the companion document TRYTEMPL. You can also)LOAD KEYS after you are in TryAPL2 to cause a display of an APL2 keyboard and rearrange the keyboard for non-US configurations. Notice that, when you are using APL2, the primary alphabet is uppercase. Pressing an alphabetic key will give you an uppercase character. A lowercase letter is entered by pressing "alt" and an alphabetic key. Pressing "shift" and an alphabetic key gives you the APL2 special symbol inscribed on the upper part of the keytop. For the non-alphabetic keys, pressing the key gives you the character inscribed on the lower left part of the keytop. Pressing "shift" and a non-alphabetic key gives you the character inscribed on the upper part of the keytop. Pressing "alt" and a non-alphabetic key gives you the character inscribed on the lower right part of the keytop. On many terminals, the "alt" characters are inscribed on the front face of the key rather than on the top.

If the APL2 characters do not appear on your screen, try pressing alt-F8 to switch into graphics mode.

To see APL2 characters when running in a DOS full screen window in OS/2, your CONFIG.SYS should contain the following line:

```
DEVINFO=SCR.BGA,C:\TRYAPL2\APL2VIO.DCP
```

Adjust the path information if you have not installed TryAPL2 in the directory TRYAPL2.

Installing TryAPL2

Before using TryAPL2, make a backup copy of the distribution diskette. If your computer has a fixed disk, you may want to copy the distribution disk to your fixed disk. (Modify these instructions to meet your own needs.):

```
To install TryAPL2 to a hard disk:
  - create a directory on the hard disk. For example:
            md tryapl2
  - make your new directory the current directory. For example:
            cd tryapl2
  - insert the TryAPL2 disk in drive A and execute the following:
            copy a:contract
            copy a:read.me
            a:tryapl2a
            a:tryapl2b
            a:tryapl2c
  - if you want to restore the LaserJet formatted documentation
    (on disk 2 of the 5.25" set), insert the TryAPL2 disk
    in drive A and execute the following:
            a:tryapl2d
```

```
To install TryAPL2 on a floppy disk:

  - make the floppy drive the current drive. For example:
            b:
  - insert the TryAPL2 disk in drive A and execute the following:
            a:tryapl2a
            a:tryapl2b
            a:tryapl2c

  NOTE: If you install to 360K disks, you must change disks between
        the above steps.

  NOTE: The files from tryapl2d are documentation for printing
        on HP LaserJet printers, are very large, and are not required
        for execution.

  NOTE: Do not mark any files with the readonly attribute.


To run TryAPL2 from a hard disk:

  - Make active the directory containing TryAPL2 and enter the following:
             tryapl2

To run Tryapl2 from a floppy disk,

  - insert the first or only TryAPL2 disk that you created into drive A
    and execute the following:
             tryapl2

  NOTE: You can run from a single 360K disk using the files on
        tryapl2a. You will need to change the disk to tryapl2b
        after TryAPL2 is running to access workspaces other
        than TRYDOC. To run TRYDOC, )LOAD with disk 1 inserted
        then switch to disk 3 before selecting a topic for viewing.
```

Printing the TryAPL2 documentation

Two documentation files are provided in TRYAPL2D.EXE. You can print on LaserJet type printers. They are:

- TRYAPL2D.HPC - printable form of the information in the TRYDOC workspace
- TRYTEMPL.HPC - keyboard templates with APL2 characters for some popular keyboards

These files are very large when uncompressed but you can send them to a printer directly without decompressing them. To print the documentation on printer LPT1, use:

```
tryapl2d tryapl2d.hpc -pb1
tryapl2d trytempl.hpc -pb1
```

If you get a "not ready" error during this printout, either use the MODE LPT1:80,6,P command (for DOS 3.3) or the MODE LPT1 RETRIES=B command (for DOS 4 or DOS 5), and start the print again.

Getting in and out and around TryAPL2

Set your default drive to the drive that contains TryAPL2:

```
c:
```

If you have installed TryAPL2 on your hard disk, switch to the proper directory:

```
cd \tryapl2
```

Begin execution by typing:

```
tryapl2
```

(If you are using a national codepage, begin execution by typing "tryapl2n." See section on national codepages.)

After a moment for initialization, you will find yourself in a simple full screen APL2 application that lets you view the online documentation. You may scroll through the topics using the "Page Up" and "Page Down" keys. Pressing the space bar on any topic selects it and pressing enter shows you the contents of the selected topics. If '[*MORE*]' is displayed on the bottom line of the screen, the topic is larger than one screen and scrolling is required to see everything. Press "Esc" to exit this workspace. You may reenter the documentation workspace again by entering:

```
      )LOAD TRYDOC
```

When you exit from *TRYDOC*, you are in the APL2 session manager in an active workspace area denoted by *CLEAR WS* . What you type and the responses are recorded in a log file that is retained across sessions. The cursor keys and the "Page Up" and "Page Down" keys may be used to scroll forward and backwards in the log. If you position the cursor on a line and press enter, that line becomes the first line on the screen.

Try some of the examples that follow to see if TryAPL2 is working properly on your system.

Things you type are normally indented six spaces from the left margin. Responses from TryAPL2 are normally against the left margin. For example type the first of the following lines and press enter:

```
      +/10 20 30
60
```

If you make a mistake, an error is reported. In the full APL2 product you would need to type a right arrow to clear the error. TryAPL2, however, automatically clears the error and displays the right arrow as though you had typed it:

```
      2+
SYNTAX ERROR
      2+
        ^
      →
```

You may correct an error by moving the cursor up and typing on top of any line on the screen. When you press enter, the altered line is copied to next available line on the screen and re-executed. If you modify more than one line, they are copied one at a time from top to bottom and re-executed.

You can move to another workspace and make it the active workspace by using the system command *)LOAD*. Try this by loading the supplied workspace *CALENDAR*:

```
      )LOAD CALENDAR
SAVED 1989-08-01  07:41:59 CALENDAR
```

The save date you see may be different from the one shown here and there will be some descriptive information given.

The *CALENDAR* workspace contains a program called *JD* which converts a Gregorian date to a Julian day number. For example, here is the Julian day number for August 1, 1989:

```
      JD 1989 8 1
2447740
```

Julian day numbers are useful for doing date arithmetic. For example, if you want to know how many days between Bastille day and the opening of the Woodstock rock concert, you enter:

```
      (JD 1969 8 17)-JD 1789 7 14
65777
```

Find out how many days you've been alive by putting today's date on the left and your birth date on the right in the above expression.

If you ever start a program and want to stop it before it finishes, press the break key.

Now throw away your active copy of the *CALENDAR* workspace by entering the following:

```
      )CLEAR
```

The copy of the *CALENDAR* workspace on your disk is not affected by this operation.

When you are finished using TryAPL2, you leave by entering:

```
      )OFF
```

You'll be prompted to press "Enter" one more time before TryAPL2 returns to DOS.

APL2 Workspaces

The unit of APL2 memory is the workspace. Everything you do is remembered in a workspace. Normally when you begin an APL2 session, you are in an unnamed active workspace called *CLEAR WS*. This means that there is no memory of any variables or programs. You can use system commands to work with collections of APL2 programs and data provided in saved workspaces that you have created or are provided with TryAPL2. To see what workspaces are available, use the *)LIB* command:

```
      )LIB
CALENDAR DISPLAY GRAPHS   IDIOMS  IE       KEYS     OR
PRINT    SEARCH   STATS   TRYDOC
```

Note that the *)LIB* command does not list the workspace *CLEANSPACE* which is the first saved APL workspace and is included for historical reasons only. You'll learn about these workspaces in the chapter on Supplied Workspaces.

System Commands

To get in and around and out of the workspaces you use system commands -- lines that start with the ")" symbol. You have already used the)*LOAD* command to activate the saved *CALENDAR* workspace and the)*CLEAR* command to throw away a workspace. Other useful system commands are shown by example next. First, activate the *OR* workspace so some programs and data are available:

```
      )LOAD OR
SAVED 1989-07-21  8.02.01
```

The date and time that the workspace was saved is given. The date you see may be different.

To see what is in this workspace, use the)*NMS* command:

```
      )NMS
ARCS.3  DESCRIBE.2      PATHSFROM.3      SETUP.3 SPM.2    VALUE.3
```

A name with a .2 after it is the name of a variable. To see its value, enter its name:

```
      SPM
0  9  14  0   0   0   0
0  0   6  7  11   0   0
0  0   0  2   0  19   0
0  0   0  0  16   8   0
0  0   0  0   0   0  20
0  0   0  0  12   0  11
0  0   0  0   0   0   0
```

A name with a .3 after it is the name of a user-defined function. A name with a .4 after it is the name of a user-defined operator. In the *OR* workspace there are three user-defined functions and no user-defined operators. You will see how to display both of these in the chapter on APL2 language in the defining and editing section.

To see just a listing of the variables in your workspace use the)*VARS* command:

```
      )VARS
DESCRIBE   SPM
```

Similar commands,)*FNS* and)*OPS*, display individual lists of the functions and operators.

To get rid of an object in the active workspace use the)*ERASE* command:

```
      )ERASE SPM
```

You can give a name to the active workspace (or change its name) using the)*WSID* command (Workspace Identification):

```
      )WSID MYWS
WAS CLEAR WS
```

If you have defined some programs and data that you want to save, use the)*SAVE* command:

```
      )SAVE
SAVED 1989-08-23  9.12.01  MYWS
```

This places a copy of the objects in the active workspace into a file. If the workspace has not been given a name, you may include the name with the command:

```
      )SAVE MYWS
SAVED 1989-08-23  9.12.01
```

The)*COPY* command selects objects from a saved workspace and adds them to the objects in the active workspace. For example, the following command gets the *SPM* variable from the *OR* workspace and puts it into the active workspace:

```
      )COPY OR SPM
SAVED 1989-07-21  8.02.01
```

You get rid of a saved workspace using the)*DROP* command:

```
      )DROP MYWS
1989-08-23  9.12.01
```

The APL2 Language

The following language summary introduces the main ideas of APL2 programming. It is not expected that you can learn to program in APL2 given only this information. It is intended to cover the main ideas so you can begin to discover and explore what the language can do for you. Such a short summary can give you only a limited appreciation of the power of APL2. Complete information is available with the documentation that comes with the full product and from other books referenced in this document. You can also learn more by trying the lessons in the appendix and by studying the APL2 programs in the example workspaces supplied with TryAPL2.

APL2 data

APL2 supports two types of data - numbers and characters. A number is entered as in the following examples:

```
1234      an integer
 1.4      a fractional number
 ¯45      a negative number
1.2E11    a number in scientific notation
```

The language makes no distinction between integer, floating-point, or logical data. The language does distinguish between the symbols for negative (¯) and subtraction (-).

A character is entered between single quotes:

```
'W'
```

On output, quotes on character data are not displayed.

APL2 is best at computing on collections of data called *arrays*. The simplest type of array consists of a single number or a single character and is called a *simple scalar*.

An array can be a list in which case it is called a *vector*. For example, here is a vector of numbers of length 3, a vector of characters of length 4, and a mixed vector of length 5:

```
10 2.1 0
'F' 'A' 'T' 'E'
10 'A' 2.1 'E' 0
```

When a vector contains only items that are single characters (simple scalars), it may be written more compactly. The following are the same character vector:

```
'F' 'A' 'T' 'E'
'FATE'
```

More varied collections of data are produced by various APL2 functions.

APL2 functions

APL2 primitive operations are represented by symbols. Operations on data are called *functions*. A function may take one argument on the right (a *monadic* function) or an argument on the left and on the right (a *dyadic* function). A system-supplied function is called a *primitive function* and is represented by a symbol, a user-supplied function is represented by a word.

Here is an example of the **reciprocal** (monadic ÷) primitive function:

```
      ÷ 2 4 .01
0.5 0.25 100
```

Here is an example of the **multiply** (dyadic ×) primitive function:

```
      100 × .05 .06 .09
5 6 9
```

Here is an example of a user-defined function:

```
      ROOT 16 100 2
4 10 1.414213562
```

Here is an example of the **interval** (monadic ⍳) primitive function:

```
      ⍳10
1 2 3 4 5 6 7 8 9 10
```

Notice that in the first three examples the arguments are vectors of length 3 and that APL2 can operate on all elements of the vector at once. This can be a powerful tool as you will see later.

A single function (symbol or word) can have two meanings ... monadic and dyadic. Here is an example of a **subtraction** (dyadic -) function and a **negation** (monadic -) function:

```
      2 - 8
6
      -5 7 ¯2
¯5 ¯7 2
```

Functions are often used in combination with other functions. Here is an example of a user-defined function combined with the **power** function and the **multiplication** function combined with the **addition** function:

```
      (4*2) + ROOT(3*2)
19
      1.05 × 10 + 20
31.5
```

This last combination brings up an interesting feature of APL2. This evaluation is different from normal arithmetic where you multiply before you add regardless of the order of the functions in the expression. APL2 has over 80 functions and remembering which function is executed before another would be nearly impossible. Therefore, APL2 has an easy rule: ***Execute functions from right to left.*** This means that all functions have equal precedence. Parentheses can be used to group data and functions to control the order of execution.

Functions are also provided to rearrange data. Here are examples of the **reshape** (dyadic) function:

```
      8⍴'FATE'
FATEFATE
      3⍴0
0 0 0
      3 2⍴ 7 11
7 11
7 11
7 11
```

The last example shows that arrays need not be linear. A two-dimensional array is called a *matrix*. Higher dimensional arrays are also allowed. Here is an example of a three-dimensional array:

```
      2 3 4⍴⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
```

Notice that this array displays as a two-dimensional array separated by a blank line. Although it is necessary to display them this way, it is easier to think of them as behind one another like pages in a book.

In arrays of two dimensions or more, the rightmost axis is called *columns* , the second from the rightmost axis is called *rows* , and the third from the right is usually called *planes.* In arrays higher than three, the other axes are not normally given names but are sometimes called *hyperplanes* .

An array is given a name by using a left arrow. Here is an example of a simple scalar array, a vector array, and a two-dimensional array being given the names *SS*, *V*, and *DD* respectively:

```
      SS←'K'
      SS
K
      V← 2 4 6
      V
2 4 6
      DD←2 3⍴⍳6
      DD
1 2 3
4 5 6
```

A name which has an array as its value is called a *variable.* Notice that mentioning the name of a variable elicits its value.

Names must begin with an alphabetic symbol or one of the special symbols Δ or ⍙. Other symbols in a name may be these characters or the digits 0 to 9. APL2 implementations normally support two alphabets (uppercase and lowercase or uppercase and uppercase underscored). In this document, uppercase and lowercase characters are used in names. TryAPL2 only permits names of seventeen characters or shorter to be saved in a workspace. The full product has no such restriction.

The **shape** (monadic ⍴) function, when applied to a vector, tells you how many items are in the array. Here are examples:

```
      ⍴'FATE'
4
      ⍴10 20 30
3
```

The **shape** function, when applied to a higher dimensional array, tells you how long each of the axes are; that is, how many rows, columns, etc. It returns a integer vector as shown by this example:

```
      ⍴DD
2 3
```

The **shape** of the **shape** of an array, says you how many numbers are used to describe the shape. This is, by definition, the *rank* of an array. Here are some examples:

```
      ⍴⍴'FATE'
1
      ⍴⍴10 20 30
1
      ⍴⍴DD
2
```

Thus, *DD* has shape 2 3 and rank 2.

APL2 has special names for arrays that have rank 0, 1, and 2:

- Rank 0 - scalar
- Rank 1 - vector
- Rank 2 - matrix

All the arrays you've seen so far have had as items single numbers and single characters. In APL2 an item of an array can be any arbitrary array. Here is a three-item vector whose first item is a scalar, second item is a vector, and third item is a matrix:

```
      N←3 'ABC' (2 3⍴⍳6)
      ⍴N
3
```

An array where at least one item is other than a single number or a single character is called a *nested array*.

When a nested array is displayed, APL2 uses blanks to indicate structure:

```
      N
 3 ABC   1 2 3
         4 5 6
```

An unambiguous presentation of the structure of a nested array is produced by the *DISPLAY* function from the *DISPLAY* workspace:

```
      )COPY DISPLAY DISPLAY
SAVED 1989-07-21  8.02.01

      DISPLAY N
┌→────────────────┐
│  ┌→──┐ ┌→────┐ │
│ 3│ABC│ ↓1 2 3│ │
│  └───┘ │4 5 6│ │
│        └~────┘ │
└∊────────────────┘
```

The **depth** function tells you how far into the array you need to go to find the deepest simple scalar:

```
      ≡N
2
```

Depth is easy to compute using the output of the *DISPLAY* function. Count the number of boxes you enter when drawing a line from outside the display to each number or character in the display. The largest number you count matches the depth of the array.

Arrays of depth 0 and 1 are called *simple arrays*. Arrays of depth 2 or more are called *nested arrays*.

APL2 scalar functions

The *scalar functions* are those defined on single numbers or characters that extend to array arguments in a uniform way. They include functions for simple arithmetic, logarithms, trigonometry, and logic. You have already seen a few examples in the previous section. The "Summary of APL2 Functions and Operators" gives a complete list of the primitive scalar functions.

The monadic scalar functions return a result that has the same structure as the argument. Here are examples with the functions **negate**, **reciprocal**, and **floor**:

```
      -2 3⍴⍳6
¯1 ¯2 ¯3
¯4 ¯5 ¯6

      ÷1 2 10
1 0.5 0.1

      ⌊ 2.3 45 ¯2.3
2 45 ¯3
```

The dyadic scalar functions apply between corresponding items one from each argument:

```
      10 20 30+1 2 3
11 22 33
```

This is the same as:

```
      (10+1)(20+2)(30+3)
11 22 33
```

If one argument is a scalar, it is paired with each item of the other argument:

```
      10 20 30-1
9 19 29
```

When applied to nested arrays, these rules are applied recursively:

```
      (2 3) 4 (5 6 7)+10 20 (30 40 50)
 12 13  24  35 46 57
```

This is the same as:

```
      (2 3+10)(4+20)(5 6 7+30 40 50)
 12 13  24  35 46 57
```

The monadic **circular** function produces products of pi:

```
      ○1 1.57079 2
3.141592654 4.934782324 6.283185307
```

APL2 primitives assume that angles are expressed in radians. Since a circle has two pi radians or 360 degrees, the following expression converts degrees to radians. Here is one degree expressed as radians:

```
      ○2÷360
0.01745329252
```

The dyadic **circular** function provides a set of mathematical functions including trigonometric, hyperbolic, and arc functions. Here are examples of **sine**, **cosine**, and **tangent** of various angles:

```
      1○30 45 90×○2÷360
0.5 0.7071067812 1

      2○30 45 90×○2÷360
0.8660254038 0.7071067812 2.832769449E¯16

      3○30 45    ×○2÷360
0.5773502692 1
```

Logical operations are defined in APL2 using the numbers `1` and `0` as the logical values true and false. The **and** function produces true only when both its arguments are true. Here are all four possible combinations of true and false with the function **and**:

```
      0 0 1 1 ∧ 0 1 0 1
0 0 0 1
```

Here are examples of **or**, **nor**, and **not equal** (also called **exclusive or**):

```
      0 0 1 1 ∨ 0 1 0 1
0 1 1 1

      0 0 1 1 ⍱ 0 1 0 1
1 0 0 0

      0 0 1 1 ≠ 0 1 0 1
0 1 1 0
```

APL2 mixed functions

Primitive functions that are not scalar functions are called *mixed functions*. You've already seen **shape**, **reshape**, and **depth**. The "Summary of APL2 Functions and Operators" gives a complete list of the primitive mixed functions.

Enclose adds levels of nesting to an array. Here are examples of **enclose** turning a matrix into a scalar:

```
      A← 2 3⍴⍳6
      ⊂A
 1 2 3
 4 5 6
      ⍴⊂A     The shape of a scalar is an empty vector
              (an empty vector prints as a blank line)
      ⍴⍴⊂A    A scalar is a rank-zero array
0
      ≡⊂A
2
```

The *DISPLAY* of a nested scalar consists of a box with no arrows because a scalar has no shape (more precisely, a scalar has empty shape). Inside the box is the display of the item of the scalar:

```
      DISPLAY⊂A←2 3⍴⍳6
┌∊─────────┐
│ ┌→────┐  │
│ ↓1 2 3│  │
│ │4 5 6│  │
│ └~────┘  │
└─────────┘
```

Here is an example of **enclose** along an axis used to turn a matrix into a vector of its rows:

```
      ⊂[2]A
 1  2  3   4  5  6
      ⍴⊂[2]A
2
```

Disclose (⊃) is the inverse of **enclose**.

Index is a function that selects cross-sectional subsets of an array. There are two ways to denote the function; with the squad function symbol [] and with special non-functional syntax using a matched pair of square brackets. Here are examples of selecting specified rows (1, 3, and 5) and columns (4 and 5) from a matrix using each of these denotations:

```
      A←8 8⍴⍳64
      A[1 3 5;4 5]
 4  5
20 21
36 37
      (1 3 5) (4 5)[] A
 4  5
20 21
36 37
```

Brackets are an older and more familiar notation. Squad **index** is a function without special syntax and so can be used with operators.

A rank-N array is indexed with N index arrays when using brackets and with an N-item vector when using squad. A scalar may be used in place of a one-item vector. Here are examples of selection from a vector:

```
      A←'ABCDEFGHIJ'
      A[4]
D
      4[]A
D
      A[3 1 4 4 5]
CADDE
      (⊂3 1 4 4 5)[]A
CADDE
```

Notice the use of **enclose** in the last example. Since a vector is a rank-one array, it must be indexed with a length-one array. Since a scalar can be used in place of a length-one array, **enclose** can be used to produce the scalar index. The vector `3 1 4 4 5` would be suitable to select one item from a rank-five array.

You can use **index** to replace items:

```
      A←'ABCDEFGHIJ'
      A[4 6]←'XZ'
      A
ABCXEZGHIJ
```

This operation is called a *selective assignment*. Notice that nothing is printed in response to any assignment operation. The result returned by an assignment is the array on the right, not the array in which the replacement occurred. Even though it doesn't print, it can be used in further computations:

```
      A←'ABCDEFGHIJ'
      4⍴A[4 6]←'XZ'
XZXZ
```

There are many other functions that select subsets of arrays. **Take** and **drop** select by specifying the number of items to be kept or discarded from an array:

```
      3↑'ABCDEFG'
ABC
      3↓'ABCDEFG'
DEFG
```

A negative number causes the operation to be applied to the right side of the array:

```
      ¯3↑'ABCDEFG'
EFG
      ¯3↓'ABCDEFG'
ABCD
```

On a higher-rank array, you may specify on the left one number per dimension:

```
      3 ¯2↑8 8⍴⍳64
 7  8
15 16
23 24
```

You may also specify application along an axis if you want to select all of one axis but only part of another:

```
      3↑[1]8 8⍴⍳64
 1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
```

Replicate (also called **Compress**) uses a left operand to say how many copies to make of the corresponding item from the right:

```
      1 0 1 0 1/⍳5
1 3 5
      ⍴1 0 1 0 1/⍳5
3
      2 1 2 1 2/⍳5
1 1 2 3 3 4 5 5
```

More mixed functions

First selects the first item of an array:

```
      ↑10 20 30
10
      ↑(2 3)(4 5)(6 7)
2 3
      ⍴↑(2 3)(4 5)(6 7)
2
```

Notice the distinction between **first** and **take**:

```
      DISPLAY ↑(2 3)(4 5)(6 7)
┌→──┐
│2 3│
└~──┘
      DISPLAY 1↑(2 3)(4 5)(6 7)
┌→──────┐
│ ┌→──┐ │
│ │2 3│ │
│ └~──┘ │
└∊──────┘
```

Take with a left argument of 1 selects a 1-item vector that contains the first item of the right argument. **First** selects the first item which can be any arbitrary shape.

First and **drop** are the APL2 equivalents of CAR and CDR in LISP.

Pick selects an item at an arbitrary depth in an array. The length of the left argument determines at what depth the selection occurs. Here are examples of **pick** applied to a depth-three array:

```
      A←((2 3)(4 5))('ABCD' 'EFGH')
      (⍳0)⊃A                ⍝ empty PICK selects all
  2 3  4 5  ABCD EFGH
      1⊃A
 2 3  4 5
      1 2⊃A
4 5
      1 2 1⊃A
4
```

Notice that an empty pick selects the entire array and that an argument of length N selects a simple scalar from an array of depth N.

Catenate joins the values of two arrays:

```
      A←2 3 4
      B←'ABCD'
      A,B
2 3 4 ABCD
      ⍴A,B
7
```

Given two rank-two arrays having the same number of rows, **catenate** joins the columns:

```
      A←2 3⍴⍳6
      B← 2 4⍴10 20 30 40 1 2 3 4
      A,B
1 2 3 10 20 30 40
4 5 6  1  2  3  4
      ⍴A,B
2 7
```

You can join on the rows by specifying the first axis:

```
      A←2 3⍴⍳6
      C←3 3⍴'ABCDEFGHI'
      A,[1]C
1 2 3
4 5 6
A B C
D E F
G H I
```

Match returns 1 or 0 depending on whether its two arguments are the same in value and structure or are different:

```
      2 3 4 ≡ 2 3
0
      2 3 4 ≡ 1+1 2 3
1
      (2 3)(4 5) ≡ (2 3)(6 7)
0
```

Matrix inverse computes the algebraic inverse of a matrix:

```
      I←3 3⍴14 ¯140 168 ¯40 640 ¯840 27 ¯540 756
      I
 14 ¯140  168
¯40  640 ¯840
 27 ¯540  756

      ⌹I
0.3333333333  0.1666666667  0.1111111111
0.08333333333 0.06666666667 0.05555555556
0.04761904762 0.04166666667 0.03703703704
```

The inverse of a matrix has the property that when combined with the original matrix by an algebraic inner product (described later) an identity matrix is produced:

```
      (⌹I)+.× I
1 0 0
0 1 0
0 0 1
```

Because of numerical approximations, the answer you see may not be a perfect identity matrix.

Matrix divide may be used to find a solution to a set of simultaneous linear equations such as these:

$$\begin{aligned} X+\ Y+Z &= 3 \\ 4X+2Y+Z &= 8 \\ 9X+3Y+Z &= 15 \end{aligned}$$

These equations are satisfied when X is 1, Y is 2, and Z is 0. Here is the computation of these roots where *SE* is the matrix of coefficients:

```
      SE
1 1 1
4 2 1
9 3 1

      3 8 15⌹SE
1 2 ¯4.439807896E¯16
```

APL2 operators

APL2 makes a distinction between functions and operators. *Functions* are operations that apply to data and produce new data. *Operators* are operations that apply to functions and produce new functions.

Reduction (/) applies to a dyadic function and produces a related monadic function. For example, **reduction** applied to the **addition** function produces the **summation** function:

```
      +/10 30 20
60
```

Reduction can be applied to any dyadic function including programs that you write yourself. Here are examples with the functions **maximum** and **catenate**:

```
      ⌈/10 30 20
30
      ,/2 3⍴10 30 20 40 50 60
 10 30 20  40 50 60

      DISPLAY ,/2 3⍴ 10 30 20 40 50 60
┌→──────────────────────┐
│ ┌→──────┐ ┌→──────┐  │
│ │10 30 20│ │40 50 60│  │
│ └~──────┘ └~──────┘  │
└⊂──────────────────────┘
```

N-wise reduction is like **reduction** except that a set of reductions are performed on overlapping sections of the right argument. A number on the left says how many items participate in each reduction. Here is an example of a pairwise difference:

```
      2-/10 20 30 50 70
¯10 ¯10 ¯20 ¯20
```

A negative number on the left causes each vector to be reversed before it is reduced. Here is a pairwise difference that in mathematics is called a first difference:

```
      ¯2-/10 20 30 50 70
10 10 20 20
```

Scan (\) is like a set of reductions:

```
      +\10 30 20
10 40 60
    is like
      (+/10)(+/10 30)(+/10 30 20)
10 40 60
```

Scan can be applied to any dyadic function including programs that you write yourself.

Each applied to a monadic function applies the function to each item of its argument:

```
      ⍴¨(2 3)(4 5 6)(2 3⍴⍳6)
 2  3  2 3
    is like
      (⍴2 3)(⍴4 5 6)(⍴2 3⍴⍳6)
 2  3  2 3
```

Each applied to a dyadic function applies the function between corresponding items one from each argument:

```
      (2 3)(4 5)≡¨(2 3)(6 7)
1 0
    is like
      (2 3≡2 3)(4 5≡6 7)
1 0
```

If one argument is a scalar, then it is paired with each item from the other argument:

```
      2⍴¨5 6 7
 5 5  6 6  7 7
```

Each may be applied to any function including programs you write yourself.

Outer product (∘.) produces a function that applies between all combinations of items, one from each argument:

```
      10 30 20 ∘.+ 1 2
11 12
31 32
21 22
      ⍴10 30 20 ∘.+ 1 2
3 2
```

The "jot" (∘) symbol is just a place-holder and is not a function. "Jot" is only used in **outer product**. Notice that the result shape is derived by adjoining the shapes of the arguments. The words "all combinations" in a problem can often be translated to "**outer product**." **Outer product** can be applied to any dyadic function including one you write yourself. Here's **outer product** applied to **catenate**:

```
      10 30 20 ∘., 1 2
 10 1  10 2
 30 1  30 2
 20 1  20 2
```

This result is a 3 by 2 matrix of pairs.

Inner product combines **reduction** and **outer product** in one operation:

```
      D← 2 3⍴⍳6
      E← 3 4⍴⍳12
      D+.×E
38 44  50  56
83 98 113 128
      ⍴D+.×E
2 4
```

Rows from the left argument are multiplied by columns from the right argument in all combinations and the resulting vectors are reduced with plus. **Inner product** with functions **addition** and **multiplication** is the APL2 notation for the algebraic inner product. Any dyadic function may be substituted for + and × including programs that you write yourself.

APL2 errors

If you make a mistake, APL2 will produce an error message and show you the line that contained the error. A caret (^) shows you how far APL2 got in its right to left scan. Some APL2 implementations show a second caret to the right to show which operation failed.

APL2 error messages are very brief. However, they are also few so figuring out what caused an error is usually not difficult. When an error occurs in the middle of an expression, keep in mind that everything to the right of the caret has been looked at by APL2 and executed if possible. Also remember that TryAPL2 automatically clears errors with a right arrow. This keeps the workspace clean but can make debugging more difficult.

Here are some expressions that contain errors:

- RANK ERROR - argument ranks are incorrect. For example, dyadic scalar functions must have arguments of the same rank unless one of them is a scalar:

```
      (2 3⍴⍳6)+(2 3 4⍴⍳12)
RANK ERROR
      (2 3⍴⍳6)+(2 3 4⍴⍳12)
      ^
```

- LENGTH ERROR - argument shapes are incorrect. For example, dyadic scalar functions must have arguments that have the same lengths unless one of them is a scalar:

```
      2 3+4 5 6
LENGTH ERROR
      2 3+4 5 6
      ^
```

- DOMAIN ERROR - argument values are not appropriate for the function or the result cannot be represented:

```
      2 3+'A'
DOMAIN ERROR
      2 3+'A'
      ^
```

- SYNTAX ERROR - APL2 could not figure out what the expression means:

```
      2+
SYNTAX ERROR
      2+
      ^
```

- INDEX ERROR - an index did not select an item from an array:

```
      A← 2 3 4
      A[6]
INDEX ERROR
      A[6]
      ^
```

Defining and editing your own programs

You can write programs that act like data, functions, or operators. Such programs are called *defined*. Your programs have the same properties as primitive data, functions, or operators. They may be monadic or dyadic, participate in APL2 expressions, and functions may be applied with operators (primitive or defined).

Real APL2 systems contain full-function full-screen editors and allow use of standard system editors. All APL systems have a simple line editor called the "del" editor because of the symbol used to invoke it. TryAPL2 contains only a subset of this editor. A few basic functions of the "del" editor are shown here. See the documentation that comes with the full APL2 product for complete details. Even though the "del" editor is very simple, the session manager gives it many of the features of a full screen editor. A good way to use the editor on an existing function is to display a set of lines on the screen and use the session manager to make whatever changes are needed. You enter the editor for a new program by typing a "del" (∇) followed by the header of the program. Here is an example:

```
      ∇Z←F C;Y
[1]
```

The header gives all the information about how the program can be called and what result it returns. In this example, the name of the function is *F*. It can be called with one argument on the right so it is monadic. Inside the program, this argument will be referred to by the name *C*. Inside the program, a variable named *Z* will be produced and this value is to become the result of the program. *Y* is a name that is used inside the program for some temporary purpose which has no meaning outside the program. Such a name is called a *local name*. Except for the name of the function itself, all names in the header are local names. They have meaning only inside the program and do not conflict with other objects that have the same name outside the program.

You enter the APL2 statements which make up the program one at a time as the editor prompts you for new lines:

```
      ∇Z←F C;Y
[1]   Z←32+1.8×C
[2]
```

You get out of the editor by typing "del" at the end of a line:

```
      ∇Z←F C;Y
[1]   Z←32+1.8×C
[2]   ∇
```

You run the program by entering its name and giving it an argument:

```
      F 0 20 37 100
32 68 98.6 212
```

Note that the argument to *F* is the four item vector `0 20 37 100`. In some languages, this would be treated as four parameters. In APL2, it is a single array passed as argument to the function *F*.

The result returned by *F* is available for further computation. Notice that the functions are evaluated from right to left:

```
      459.72+F 0 20 37 100  ⍝ absolute temperatures
491.72 527.72 558.32 671.72
```

Once you have entered the editor for the first time and defined the header, you get into it again by just using the name of the program not the whole header:

```
      ∇F
[2]
```

The editor prompts you for a new line. You enter more lines at the end of the program by typing them after the line number prompts.

You can see the definition of the program by entering a quad (⎕) between square brackets after a line number prompt:

```
      ∇F
[2]   [⎕]
[0]   Z←F C;Y
[1]   Z←32+1.8×C
[2]
```

You add a line between two existing lines by using a fractional line number. Here a comment is added at the front of the program:

```
      ∇F
[2]   [⎕]
[0]   Z←F C;Y
[1]   Z←32+1.8×C
[2]   [.1] ⍝ FAHRENHEIT FROM CELSIUS
[0.2] ⍝ A SECOND COMMENT
[0.3]
```

You delete a line by entering in square brackets a "delta" followed by the line number:

```
      ∇F
[2]   [□]
[0]   Z←F C;Y
[1]   Z←32+1.8×C
[2]   [.1] ⍝ FAHRENHEIT FROM CELSIUS
[0.2] ⍝ A SECOND COMMENT
[0.3] [Δ.2]
[0.3]
```

You exit the editor by entering a "del" (∇) after a line number prompt or at the end of an added line.

```
      ∇F
[2]   [□]
[0]   Z←F C;Y
[1]   Z←32+1.8×C
[2]   [.1] ⍝ FAHRENHEIT FROM CELSIUS
[0.2] ⍝ A SECOND COMMENT
[0.3] [Δ.2]
[0.3] ∇
```

When you exit the program by entering the closing "del," the program is renumbered starting from zero:

```
      ∇F[□]
[0]   Z←F C;Y
[1]  ⍝ FAHRENHEIT FROM CELSIUS
[2]   Z←32+1.8×C
[3]
```

If the function is small, it is convenient to simply display it all and then type on top of the function to make a set of changes. When you press enter, any line altered is processed one at a time from top to bottom. If the function is larger, you may request that only a portion of it be displayed by giving a range of line numbers. For example, if you were in a large function on line 10 and wanted to see lines 50 through 60, you would enter:

```
[10]  [□50-60]
```

This was an example of a defined monadic function. The header for a defined dyadic function will have a name on the left of the function name as well as on the right. The header for defined data (also called *niladic defined function* or *defined sequence*) has neither a left nor a right argument named.

Defined operators

A defined operator is a program which can be given functions as parameters as well as data. The header for a defined operator contains a parenthesized list of two or three names in place of the function name. The name of the operator is the second name in parentheses. The other names in parentheses are the names of the operands to the operator which can be either functions or data. See the *SEARCH* workspace for an example of a defined operator.

A summary of APL2 functions and operators

This section contains summaries of the different types of primitive operations. Most of them are not discussed in this document. Please refer to the documentation that comes with purchased APL2 systems or other books for more information.

Scalar functions

Monadic		Dyadic
Conjugate	+	Add
Negative	-	Subtract
Direction	×	Multiply
Reciprocal	÷	Divide
Magnitude	|	Residue
Floor	⌊	Minimum
Ceiling	⌈	Maximum
Exponential	*	Power
Natural log	⍟	Logarithm
Pi times	○	Circular
Factorial	!	Binomial
Not	~	(not scalar)
Roll	?	(not scalar)
	∧	And
	∨	Or
	⍲	Nand
	⍱	Nor
	<	Less
	≤	Not greater
	=	Equal
	≥	Not less
	>	Greater
	≠	Not equal

The dyadic Circular function is really a set of monadic functions where an integer left argument is used to select a particular monadic function. The following table documents the left arguments:

Negative left		Positive left
`(1-R*2)*.5`	0	`(1-R*2)*.5`
Arc sine *R*	1	Sine *R*
Arc cosine *R*	2	Cosine *R*
Arc tangent *R*	3	Tangent *R*
`(¯1+R*2)*.5`	4	`(1+R*2)*.5`
Inverse hyperbolic sine *R*	5	Hyperbolic sine *R*
Inverse hyperbolic cosine *R*	6	Hyperbolic cosine *R*
Inverse hyperbolic tangent *R*	7	Hyperbolic tangent *R*
`-(¯1-R*2)*.5`	8	`(¯1-R*2)*.5`
R	9	Real *R*
+*R* (conjugate)	10	|*R*
`0J1×R`	11	Imaginary *R*
`*0J1×R`	12	Phase *R*

Note that functions 8 through 12 are not available in TryAPL2 or 16 bit APL2/PC. 32 bit APL2/PC, APL2/6000, and APL2 for S/370 S/390 all include complex arithmetic.

Mixed functions

Monadic		Dyadic
Shape	⍴	Reshape
Ravel	,	Catenate
Reverse	⌽	Rotate
Transpose	⍉	Transpose
Enclose	⊂	Partition
Disclose	⊃	Pick
	⍬	Drop
First	↑	Take
(not mixed)	~	Without
Interval	⍳	Index of
Enlist	∈	Member
Grade up	⍋	Grade up
Grade down	⍒	Grade down
(not mixed)	?	Deal
	⍷	Find
	⌷	Index
	⊤	Encode
	⊥	Decode
Matrix inverse	⌹	Matrix divide
Depth	≡	Match
Execute	⍎	
Format	⍕	Format

Additionally the symbol ⊖ represents the functions **reverse** and **rotate** along the first axis.

Operators

Monadic		Dyadic
Each	¨	
Reduce/Replicate	/	
Scan/Expand	\	
	.	Array product

Additionally the symbol ⌿ represents the operator **reduce/replicate** along the first axis and the symbol ⍀ represents the operator **scan/expand** along the first axis.

APL2 for the High School Math Classroom

This section discusses the use of APL2 in the classroom.

Why use a programming language in a mathematics classroom?

Computers have become available to the mathematics classroom. It may be one computer for the whole department to share, or it may be one to two computers in a few to many classrooms, or it may be the availability of a mathematics lab. In any case, computers have become a tool as important to the mathematics classroom as pencil, paper, and the calculator.

This has led to a multitude of software packages that instruct, remediate, calculate, graph, etc. These packages are mostly mechanical; that is, students learn by practicing. It is the programming languages that allow students to do original thinking. Programming entices students to create their own algorithms and solve problems by using their visual skills to define the problem and using their analytical skills to break down problems into smaller and smaller parts.

Programming languages teach students how to think. Mathematics comes alive as students create their own algorithms to solve problems rather than just calculate the answers.

Why use APL2?

APL2 is one of the few languages that integrates analytical and visual skills. It becomes more than a language. It becomes a way of thinking.

APL2 uses a problem-solving strategy that centers on discovering patterns in a problem and then operating on these patterns to transform them. A learning environment is created that encourages the student to discover the functions of APL2 and then explore mathematical algorithms and problems. APL2 is truly subservient to mathematics, giving immediate support to the student because:

- APL2 is a precise and concise notation for the recording of ideas.
- APL2 uses common mathematical symbols.

  APL2 uses the "×" symbol for multiplication and the "÷" symbol for division instead of the (*) or (/) symbols common in most languages.
- APL2 requires no background in computer science to get started and do something significant.
- APL2 has very few rules to memorize.

  Unlike other programming languages which evaluate the functions of an expression in a particular order, APL2 simply evaluates expressions right to left. This is the same naturally intuitive rule that the English language uses. That is to say that although the message is scanned from left to right the meaning is not clear until the last word is seen. An APL2 statement is scanned from left to right, but the result depends on evaluating everything to the right. This also follows what is inferred in the generalized mathematical notation for the evaluation of functions. For example, in the expression, $f(g(x))$, the evaluation of the function, f, depends on the evaluation of the function g. This statement

seems to be processed from right to left. It is this philosophy, and the fact that APL2 has so many functions to keep in some syntactic order, that makes this rule simple, and most importantly, leaves no room for misunderstanding.

- APL2 has a powerful interactive component where the real thinking can take place.

  This allows the student the freedom to *"guess and check"*.

- APL2 allows algorithms to be written in a fraction of the number of statements that are required in other languages.

Discovering/Teaching the Basics

Working with APL2 means *discovery* (there are over 80 system-defined functions available to explore). Whether this discovery takes place in an arithmetic class, a mathematics A class, or a college prep class, few explanations are needed for the student to learn APL2. The teacher becomes the facilitator of learning.

There are five discovery lessons offered in later sections of this document. They can be taught as a unit (40-50 minutes per lesson) in either five consecutive days or in individual enrichment days over a one- to two-month time frame. The lesson plans for each follow and include:

- Lesson goals and objectives.
- A suggested pre-lecture to explain mechanics and/or rules.
- A suggested post-lecture to summarize/confirm discoveries.

At the end of the five discoveries, the students will have learned 31 primitive functions and eight derived functions. They will be ready to seriously explore many mathematical algorithms such as:

- Rounding
- Averaging
- Even numbers
- Odd numbers
- Multiplication tables
- Factors of a number (including the number)
- Factors of a number (excluding the number)

Suggested solutions to these algorithms can be found under "APL Expressions for Some Mathematical Algorithms." Some information on what a variable is and how to `)LOAD` and `)SAVE` their work will get them started on the road to discovery in mathematics.

Lesson: Discovering APL2 -- Working With Arithmetic

This lesson can be found in Appendix A.

The goal of this lesson is to introduce APL2 and show how APL2 handles data and performs operations.

The objectives of this lesson are:

1. The student will be able to find and use the APL2 character set.

2. The student will be able to distinguish between a single scalar number and a vector.
3. The student will be able to find the pattern of applying functions to conforming vectors.
4. The student will be able to observe the solution to given examples and describe in his/her own words what the APL2 function does.

The pre-lecture should include the following:

- How to use the keyboard template -- finding the APL2 characters.

  Inclusion of the distinction between "difference" (-), which is a function and "negative" (¯), which is an attribute.

- A short discussion of the definitions.

  In a lower-level class this might include the teacher role playing a function machine and the students role playing left and right arguments. The students could guess what function the teacher is.

- A short discussion of the data representation and structure.

  The idea of "conforming" vectors should be carefully explained during this discussion. The fact that the functions in this discovery all require arguments with the same number of items is important to point out to the students. A few examples that apply vectors of various lengths would help.

Students should be encouraged to work together on the exercise and should be reminded to take care filling in both the solution and the results so they will have it all for reference later.

A post-lecture should be done before the students leave if possible. If not, it should be part of the pre-lecture of the next lesson. It should include the following:

- A means to "check" their results.

  This could be done by viewing a key or by reading/researching the chapter on APL2 Language taken from this document.

- A class and/or group discussion of their results.
- A question and answer period.

Lesson: Discovering APL2 -- Working With Vectors

This lesson can be found in Appendix B.

The goal of this lesson is to explain the fundamental array structure of APL2 and explore several operations that manipulate vectors.

The objectives of this lesson are:

1. The student will be able to distinguish between a numeric vector and a character vector (string).
2. The student will be able to use functions that produce results that do not conform to the original arguments.
3. The student will be able to select various items from a vector.
4. The student will be able to observe the solution to given examples and describe in his/her own words what the APL2 function does.

The pre-lecture should include the following:

- A short discussion of what character data is and how it is represented.
- A short discussion of non-scalar functions; that is, how the solution and the arguments are NOT conforming.

The post-lecture should include the following information:

- A means to "check" their results.
- A class and/or group discussion of their results.
- A discussion of the importance of these functions in the real world; that is, in data processing.
- A question and answer period.

Lesson: Discovering APL2 -- Working With Arrays

This lesson can be found in Appendix C.

The goal of this lesson is to apply functions to two-dimensional arrays.

The objectives of this lesson are:

1. The student will be able to distinguish between a vector (one-dimensional array) and a matrix (two-dimensional array)
2. The student will be able to use functions that manipulate vectors into matrix arrangements.
3. The student will be able to perform addition, subtraction, multiplication, and division with a scalar and a matrix.
4. The student will be able to observe the solution to given examples and describe in his/her own words what the APL2 function does.

The pre-lecture should include the following:

- A short discussion of what an array is and what some of the characteristics are.

The post-lecture should include the following information:

- A means to "check" their results.
- A class and/or group discussion of their results.
- A discussion of the importance of 2-dimensional arrays in their studies in mathematics; how matrices are used in algebra.
- A question and answer period.

Lesson: Discovering APL2 -- Working With Comparisons

This lesson can be found in Appendix D.

The goal of this lesson is to introduce operations for comparing data using Boolean data and functions.

The objectives of this lesson are:

1. The student will be able to equate a numeric "1" with a value of true and a numeric "0" with a value of false.
2. The student will be able to distinguish between a vector of numeric or character data and a vector of Boolean data.
3. The student will be able to distinguish between the four possible combinations of true and false.
4. The student will be able to find the solutions to various Boolean functions and realize no two functions give the same result vectors.
5. The student will be able to observe the solution to given examples and describe in his/her own words what the APL2 function does.

The pre-lecture should include the following:

- A short discussion of Boolean data and functions.
- A few examples of some logic statements using one or more of the Boolean functions.

The post-lecture should include the following information:

- A means to "check" their results.
- A class and/or group discussion of their results.
- A discussion of the importance of Boolean logic and functions as related to the architecture of the hardware/software of a computer.
- A question and answer period.

Lesson: Discovering APL2 -- Working With Operators

This lesson can be found in Appendix E.

The goal of this lesson is to introduce the use of operators to control functions, creating new families of functions.

The objectives of this lesson are:

1. The student will be able to distinguish between application of a function itself and application of a function used with an operator.
2. The student will be able to use an operator to find all combinations of a function applied to data.
3. The student will be able to observe the solution to given examples and describe in his/her own words what the APL2 function does.

The pre-lecture should include the following:

- A short discussion of the definitions.
- A discussion of deriving new functions by combining them with operators.
- A discussion of the right to left rule of APL2.

  This should be carefully planned because it is a significant rule of APL2 that will affect all their discoveries after this lesson. A suggested example: `-/ 1 2 3 4 5` (Have groups of students brainstorm how to combine these numbers to get the solution 3).

The post-lecture should include the following information:

- A means to "check" their results.
- A class and/or group discussion of their results.

  A discussion of the **outer product** and how it produces tables (arrays) of information. An extension of this would look at the power of the **inner product** operator.
- A question and answer period.

Supplied Workspaces

TryAPL2 comes with twelve workspaces. *DISPLAY*, *PRINT*, *KEYS*, *CALENDAR*, *TRYDOC*, *GRAPHS*, *CLEANSPACE*, *STATS*, *SEARCH*, *OR*, *IDIOMS*, and *IE*. The workspaces are supplied as examples and you are encouraged to look at them and modify them. Keep the original diskette unmodified so you can make copies for your friends. To understand the examples given,you may need to refer to "The APL2 Language."

The DISPLAY workspace

The DISPLAY workspace contains the single function *DISPLAY* which may be used to show the structure of arrays. If you are not familiar with APL2 data structures, refer to the section "APL2 data." The function always returns a simple character array as result. For example, a vector is data organized along one direction. The *DISPLAY* of a vector shows a box with an arrow on the top edge to indicate data arranged along one axis:

```
      DISPLAY 5 6 7
.→-----.
|5 6 7|
'~-----'
```

A nested array is shown as a box with the *DISPLAY* of the items of the array inside:

```
      DISPLAY (2 3 4)( 2 3⍴⍳4) 'ABC'
.→-----------------------.
| .→-----. .→-----. .→--. |
| |2 3 4| ↓1 2 3| |ABC| |
| '~-----' |4 1 2| '---' |
|          '~-----'      |
'∈-----------------------'
```

Notice that the second item of the above vector is a matrix. A matrix is data arranged along two axes and is shown by a box with two arrows. Here is an example of **each** (¨) applied to the *DISPLAY* function:

```
      DISPLAY¨(2 3 4)( 2 3⍴⍳4) 'ABC'
.→-----.  .→-----.  .→--.
|2 3 4|  ↓1 2 3|  |ABC|
'~-----'  |4 1 2|  '---'
          '~-----'
```

The PRINT workspace

The print workspace contains functions to print APL2 objects on an appropriate printer. To print the value of a variable *A*, enter:

```
      PRINT A
```

To print the definition of a program *F*, enter:

```
      PRINT NUMBER ⎕CR 'F'
```

If you want to use the *PRINT* function with a GRAFTRAX printer, add the following line in the *PRINT* function between lines 15 and 16:

```
      [15.1]  SH←5
```

Here are other values you can assign to *SH* that may be of interest:

- *SH*←4 This sets up the *PRINT* function for use with the IBM 5182 Color Printer.
- *SH*←6 This sets up the *PRINT* function for use with the IBM 3852-2 Color Jetprinter.
- *SH*←7 By default, APL characters are turned into graphics characters so they will display on a wider variety of printers. If your printer has an APL font (for example, the IBM 5202 Quietwriter III with the "Courier Italic APL 12" electric font) setting the value 7 will turn off this automatic translation and give you higher quality printing.

The KEYS workspace

The *KEYS* workspace contains functions that split the screen in to halves horizontally and displays an APL2 keyboard on the top half. You may continue to use your APL2 session on the bottom half.

When the workspace is loaded, it automatically executes the expression *SHOWKEYS* *'US101'*. If that does not match your keyboard, you may also use *SHOWKEYS* with arguments of *'US102' 'FR102' 'GR102' 'PC' 'AT'*. If you want to remove the display of the keyboard at any time, copy and execute the *NOKEYS* function:

```
)COPY KEYS NOKEYS
NOKEYS
```

You may define non-US keyboards and messages with the following two functions. Valid arguments are: *'ENGLISH', 'FRANCAIS', 'DEUTSCH', 'ITALIANO'*

```
SETKEYS 'FRANCAIS'    ⍝ Change keyboard arrangement
SETMSG  'DEUTSCH'     ⍝ Change messages
```

Because TryAPL2 is a simulation, some messages will come out in English even when another language is selected.

The *KEYS* functions were donated by Gary Logan, IBM Boulder and Bernard Landaud, IBM France.

The CALENDAR workspace

The *CALENDAR* workspace contains functions of general use for dealing with the Gregorian calendar. A Gregorian date is represented by a three-item vector - year, month, and day. The *JD* function converts a Gregorian date to a Julian day number. For example, here is the Julian day number for August 1, 1989:

```
      JD 1989 8 1
2447740
```

Todays date can be selected from the APL2 time stamp variable ⎕*TS*

```
      ⎕TS
1989 8 1 6 45 15 237

      JD 3↑⎕TS
2447740
```

The Δ*D* function packages subtraction of Gregorian dates into a single function:

```
      1969 8 17 ΔD 1789 7 14
65777
```

Subtraction is only one of many functions that you might want to apply between dates. You would probably not want to multiply and divide dates but it is common to want to compare them. Simple comparison of dates will not work. Suppose you have two variables containing dates:

```
      D1←1989 9 1
      D2←1989 8 15
```

To find out if the first date occurs after the second date, you might enter this:

```
      D1>D2
0 1 0
```

Apparently, the answer is no, yes, and no. In fact, you want to do this computation on Julian day numbers:

```
      (JD D1)>JD D2
1
```

You could write a function like *ΔD* to package up this comparison. But there are many comparisons: greater than, greater or equal, etc. Rather than write a whole set of functions that are the same except for the function name and a single operation inside, you can write a defined operator which will apply any operation to dates. *JDO* is such an operator:

```
      D1 -JDO D2
16
      D1 >JDO D2
1
      D1 <JDO D2
0
```

It still doesn't make sense to multiply and divide dates but if you want to see what happens, you can do it:

```
      1989 9 1 ×JDO 1989 8 15
5.991541256E12
      1989 9 1 ÷JDO 1989 8 15
1.000006945
```

GD is the inverse of *JD*. Given a Julian day number, it returns the Gregorian date:

```
      GD 2447740
1989 8 1
```

The *WDNAME* function tells you the day of the week for a given Gregorian date. To find out on what day Christmas fell in 1988 enter:

```
      WDNAME 1988 12 25
Sunday
```

Today is:

```
      WDNAME 3↑□TS
Tuesday
```

(If you get a different answer, try again on Tuesday.)

The function *WEEKDAY* does the same thing as *WDNAME* except the day of the week is returned as an integer 0 through 6 representing Sunday through Saturday:

```
      WEEKDAY 1988 12 25
0
```

0 means Sunday.

WDNAME and *WEEKDAY* are only accurate after October 15, 1582.

Some accounting calendars are based on the day of the year. The function *DOY* given a Gregorian date, returns the day of the year:

```
      DOY 1989 8 1
1989 213
```

YOD is the inverse of *DOY*:

```
      YOD 1989 213
1989 8 1
```

MONTHOF produces a calendar of the month containing the day given as argument:

```
      MONTHOF 1989 8 1
  1989 Aug

  S   M   T   W   T   F   S
          1   2   3   4   5
  6   7   8   9  10  11  12
 13  14  15  16  17  18  19
 20  21  22  23  24  25  26
 27  28  29  30  31
```

IDAY and *LDAY* select a date for the day of the week given as left argument. The right argument determines the month in which the selection occurs. *IDAY* selects the first date for that day in the month. *LDAY* selects the last day for that day in the month. Here are expressions that determine the dates of the first and last Sunday in August 1989:

```
      0 IDAY  1989 8 1
1989 8 6
      0 LDAY  1989 8 1
1989 8 27
```

Here's the computation of the second Tuesday in August 1989:

```
       GD 7+JD 2 IDAY 1989 8 1
1989 8 8
```

The calendar functions were donated by Dick Conner, IBM World Trade.

The TRYDOC workspace

The *TRYDOC* workspace implements a full screen interface into online documentation for TryAPL2. You use the cursor keys or the "Page Up" and "Page Down" to locate a section you want to read. Use the space bar to select the sections you want to read. Use enter to view the selected sections. If [*MORE*] is displayed on the bottom line, then the section is larger than the screen and you can scroll. While documentation is displayed, pressing F2 opens a line for APL2 execution. The result of whatever you execute is displayed in the lower right corner of the screen to avoid covering up the documentation. Pressing enter again to clear out one expression and get ready for another. F3 returns you to the documentation screen.

When you run TryAPL2 from a single 360K 5.25″ disk, there is not room for the documentation. In this case, each selection will report "NOT FOUND".

The documentation functions were donated by Gary Logan, IBM Boulder.

The GRAPHS workspace

The *GRAPHS* workspace contains a few functions that exercise the graphics processor AP207. It is discussed here by example only. To get started, execute the function *SETUP*:

```
      SETUP
```

This will divide your screen in half horizontally giving you your APL2 session on the bottom half and a graphics field on the top half. The graphics fiels is 100 units high and 250 units wide. You can restore normal execution mode at any time by executing the *SETDOWN* function:

```
      SETDOWN
```

Assuming you have just executed *SETUP*, you may now execute graphics commands by assigning them to the variable *A*. For example, here is a command to draw a circle:

```
      A←'ARC' (65 65 10 10 0 360)
      A
0
```

The 0 returned from AP207 means the operation worked. The command means to draw an arc whose center is at 65 65, whose x and y radius is 10 10 and from 0 degrees to 360 degrees. You can play with these values to produce other circles and elipses.

The BEGAREA and ENDAREA commands surround calls where you want closed figures filled. *CIRCLE* is a function that draws one circle. Each time it is called, it changes the color to be used. If ever it chooses the same color as the background, you won't be able to see the circle. Here is how you would execute the *CIRCLE* function and have the circles filled:

```
      A←'BEGAREA' ''
      A
0
      10 CIRCLE¨ (5+5×⍳50),¨(55+35×10.4×⍳50)
      A←'ENDAREA' ''
      A
0
```

Notice that the last circle doesn't get filled until you issue ENDAREA. The expression on the right of the call to *CIRCLE* computes a vector of two-item vectors to be used for the center of circles. The Y axis of the pair depends on `10` which is the mathematical sine function.

Other calls you can make are these:

- *A*←'*CLEAR*' '*BLUE*' - clear the window to given color
- *A*←'*MOVE*' (50 50) - move to indicated point
- *A*←'*DRAW*' (60 60) - draw a line to this position.
  You can also give an n,2 matrix of points and there are other options.
- *A*←'*WRITE*' '*HELLO*' - write characters

Other calls can be used to define window attributes, colors, fonts, and images. These are discussed in the documentation that comes with the APL2 products.

The STATS workspace

This workspace contains a collection of basic statistical algorithms which fall into 4 families:

1. Univariate statistics
2. Random number generators
3. Multi-variate statistics
4. Distribution functions

In addition to their statistical role, many functions are written in a style which exploits the special features of APL2. Studying these functions will show you the value of APL2 as a language for statistical algorithms. Most of the functions are one line long. You can use them to play with data and you can modify them to play with the algorithms. The reader is assumed to be familiar with statistical concepts.

Family 1 : Univariate statistics

The functions in this family operate on a collection of numbers and produce a number which, in some sense, is a measure of the collection. Perhaps the most familiar measure of a set of numbers is its arithmetic average or *mean.* Here is a collection of numbers and the computation of their mean:

```
      TRY1
18.728 22.996 21.562 11.481 12.312 22.434 15.318 15.554 22.806

      (+/TRY1)÷⍴TRY1
18.132
```

The *STATS* workspace contains a function *MEAN* which uses a slight variation of this computation:

```
      MEAN TRY1
18.132
```

To observe the set of statistical functions provided, issue the "umbrella" function *ALL* for a collection:

```
      ALL TRY1
MEAN       18.132
SDEVN       4.5937
VARIANCE   21.102
SKEWNESS   ¯0.26474
KURTOSIS    1.4937
MAX        22.996
MIN        11.481
RANGE      11.515
MEDIAN     18.728
75 PCTILE  22.434
25 PCTILE  15.318
SEMI_IQD    7.1162
```

The examples in this section show answers to five digits precision. APL2 systems normally show answers to ten digits precision and perform calculations to even higher precision.

Each of the statistics above can be computed by itself. For example:

```
      RANGE TRY1
11.515
      75 PCTILE TRY1
22.434
```

A function *MODE* is also available. If a collection of data is multi-modal, the result of *MODE* is a vector; if there are no repeated values an empty vector is returned.

Moments about the origin are given by the function *MOM*:

```
      2 MOM TRY1
347.53
```

The functions *VARIANCE* and *SDEVN* have (n-1) as divisor; if divisor n is wanted, the function *MEANMOM* standing for "moment about the mean" may be used:

```
      2 MEANMOM TRY1
18.757
```

Family 2 : Random number generators.

This family provides a facility for constructing simulated data, and can conveniently be used for testing and applying the Family 1 functions. The function *RAND* contains a set of expressions for generating either vectors of random numbers or sample frequency distribution vectors from populations with given underlying probability distributions. The left argument of *RAND* is the number of random variables, the right argument is a character string, the first character of which is the first letter of one of the distributions listed below, followed by a space and then the parameters of the distribution separated by further spaces. For example, 50 random Normal variables coming from a population with mean 20 and standard deviation 3 are given by:

```
      50 RAND 'N'20 3
```

The available distributions are

- Boolean - argument is a number between 0 and 1, representing the proportion of 1s in the boolean vector result.
- Cumulative - argument is a discrete cumulative probability distribution, e.g. .1 .3 .9 1, and result is a sample cumulative frequency distribution.
- Exponential - argument is the mean.
- Frequency - input is a discrete probability distribution, e.g. .1 .2 .6 .1, result is a sample frequency distribution.
- Lognormal - argument is true mean and standard deviation.
- Normal - argument is mean and standard deviation.
- Sample - argument is discrete probability distribution; e.g. .1 .2 .6 .1, and result is a vector of 1s, 2s, 3s and 4s appropriately distributed.
- Uniform - argument is lower and upper limits of range.
- Weibull - argument is scale parameter followed by shape parameter which must be > 0. The mean is the scale parameter multiplied by the gamma function of `1+1÷`(shape parm).

Family 3 : Multivariate analysis.

The functions in this family provide linear regression, correlation, and principal components analysis. Chi-squared for vectors and for contingency tables is also included. There are two possible data structures for this family:

1. A matrix, rows of which correspond to observations and columns to variables.
2. A pair of vectors, one as left argument, the other as right.

For multiple regression, the leading column is taken to represent the dependent variable. If regression not through the origin is required, a column of 1s must be added, most conveniently at the right-hand end. There is an umbrella function *REGRESS* which carries out multivariate linear regression, and gives a summary output report. By using global variables *bs*, *es* and *est* in *REGRESS*, you can retain the regression coefficients, residuals and estimates for later use:

```
      TRY3
4   1 10
5   2 20
5   3 10
7   4 30
6.5 5 20
6.5 6 20
8   7 30
      REGRESS TRY3,1
 Regn. coeff(s). 0.38095  0.083333 2.8095
 Stan. error(s)  0.058321 0.01543  0.24915

 Error variance           0.053571
 Coeff of determination   0.98137

ANALYSIS OF VARIANCE :
  Regression  11.286   2  5.6429      105.33
  Residual    0.21429  4  0.053571
  TOTAL       11.5     6  1.9167
```

BS, *ES* and *EVAR* are also available as separate functions.

SEFIT and *SEPRED* give the standard errors of fit and prediction respectively. The left argument in each case is a vector of independent variable values at the point for which the standard errors are required. If regression is not through the origin this vector must include a one in the appropriate position. With the *TRY3* data above, the regression estimate and standard errors for the value vector (4,20) of the independent variables are:

```
      4 20 1+.×⎕←bs
0.38095  0.083333 2.8095
6
      4 20 1 SEFIT TRY3 ,1
0.087482
      4 20 1 SEPRED TRY3 ,1
0.24744
```

The special case of fitting a polynomial of given degree is given by *POLYFIT*. Only the first two columns of the right argument (i.e. one independent variable) are used. The result is the power coefficients in descending order. The result of fitting a cubic to the first three columns of *TRY3* is:

```
      3 POLYFIT TRY3
0.027778 ¯0.3631 1.9663 2.2857
```

The functions *COVM* and *CORM* provide covariance and correlation matrices respectively. For *COVM* the divisor is n. Edit line one if a divisor (n-1) is required.

```
      CORM TRY3
1        0.91951 0.88465
0.91951 1        0.66144
0.88465 0.66144 1
```

The correlation coefficient of two vectors as left and right arguments is given by *CORCOEF*:

```
      TRY3[;1]CORCOEF TRY3[;2]
0.91951
```

If the numbers are regarded as a contingency table, the chi-squared value is given by the function *CHISQ*:

```
      CHISQ TRY3
4.9067
```

The function *CHISQ* can also be used to deal with the special case of two vectors where the left argument is a set of "observed" values, and the right argument is the corresponding "expected" values:

```
      14 8 9 7 12 10 CHISQ 6⍴10
3.4
```

Family 4 : Standard distributions

This family of functions provides a computerized set of normal, t, chi-squared and F tables. Each of these tables can be used in two ways, either to give the percentile as a value of the corresponding statistic, or vice versa. There are two umbrella functions *PCT* and *INT* to deal with these two cases. The left argument of *PCT* is the percentile, and of *INT* the value of the statistic. The right argument is a character string, the first character of which is N, T, C, or F to identify the distribution, then, following a space, the distribution parameters in the case of T, C and F. For the F distribution the order of parameters is degrees-of-freedom numerator followed by degrees-of-freedom denominator. For example:

```
      95 PCT'N'
1.6452
      95 PCT'C 8'
15.491
      95 PCT'F 8 9'
3.2306
      2 INT'N'
0.97725
      3.23 INT'F 8 9'
0.95002
```

The functions *NPDF*, *TPDF*, *CPDF*, *FPDF* and *GPDF* return numerical values of the appropriate probability density functions (G = Gamma). The left argument (if any) is the parameter(s), and the right argument is a vector of x values:

```
      NPDF ¯3+⍳5
0.053991 0.24197 0.39894 0.24197 0.053991
      2 TPDF ¯3+⍳5
0.068041 0.19245 0.35355 0.19245 0.068041
      2 CPDF ⍳5
0.30327 0.18394 0.11157 0.067668 0.041042
```

With the exception of *NINT* the *INT*, functions use numerical integration applied to the probability density functions, and call two general purpose numerical integration functions called *SIMPSON* and *ADAPTINT* (standing for adaptive integration). These can be used quite generally. The integral of tan(x) from 0 to 1 radian can be obtained using six Simpson intervals, and then to an accuracy of five decimal places by:

```
      0 1 6 SIMPSON '3oX'
0.61582
      0 1 .000001 ADAPTINT '3oX'
0.61563
```

The statistics functions are adapted from ones donated by Norman Thomson, IBM U.K.

The SEARCH workspace

The *SEARCH* workspace contains some simple programs which play a single-person game. These programs are examples of defined operators because they take other programs as operands.

The programs *SEARCH1* and *SEARCH2* are designed to play any one-person game. You give them a start position, the desired end position, and a program that, given one position, can generate a set of next positions. The program *MOVECOLOR* generates moves for a simple colored-tile game. Suppose you have five slots in a row and you have two white tiles and two black tiles arranged in the following pattern:

```
      BEGIN
WW_BB
```

The idea of the game is to move a tile into the blank space until the following pattern is reached:

```
      END
BBWW_
```

(*BEGIN* and *END* are just five-item character vectors.)

Here is how you find the solution using *SEARCH1*:

```
      BEGIN (MOVECOLOR SEARCH1) END
 WW_BB _WWBB BWW_B B_WWB BBWW_
```

SEARCH1 generates all possible moves from each intermediate position until the end position is found. The result is the set of positions leading to the end.

SEARCH2 is similar except that you pass it an additional function that estimates how close to the end you are. Given a set of next positions to try, *SEARCH2* will try the one that is closest to the end first. *SEARCH2* is in general much more efficient. The program *ESTC* is the estimator for the color game:

```
      BEGIN (MOVECOLOR SEARCH2 ESTC) END
 WW_BB _WWBB BWW_B B_WWB BBWW_
```

This solution is the same one discovered by *SEARCH1* but that is not always true.

If you want to play some other game, you define arrays representing an initial and an ending position, define a *MOVE* function and an *EST* function for it. If you study the functions in this workspace, you may be able to write programs that solve the 15s puzzle. This puzzle is a 4 by 4 square with 15 tiles. You try to get to an end arrangement by sliding tiles into the

blank space. NOTE: for a given starting position, only half of the arrangements can be reached.

The search examples in this section are adapted from "APL2 at a Glance" by Brown, Pakin, and Polivka, Copyright 1988 Prentice-Hall Inc., ISBN 0-13-038670-7, IBM number SC26-4676, used with permission.

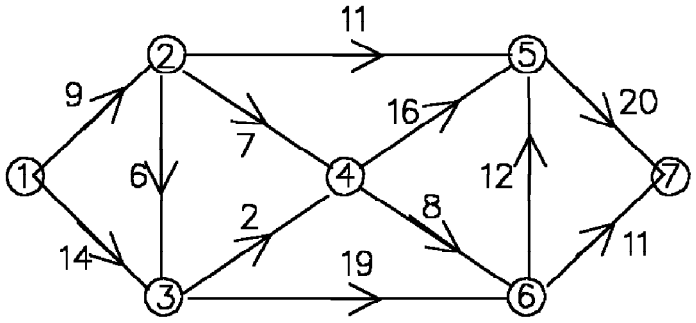
The OR workspace

The OR workspace contains functions that implement some basic algorithms of Operations Research. The functions perform operations on a directed network represented as a matrix. The functions in this workspace make use of nested arrays and recursion with **each**. If you understand these functions, you understand the basic concepts and philosophy of APL2 programming.

Suppose you have a network of six nodes connected by arcs as follows:

- from 1 to 2 with a value of 9
- from 1 to 3 with a value of 14
- from 2 to 3 with a value of 6
- from 2 to 4 with a value of 7
- from 2 to 5 with a value of 11
- from 3 to 4 with a value of 2
- from 3 to 6 with a value of 19
- from 4 to 5 with a value of 16
- from 4 to 6 with a value of 8
- from 5 to 7 with a value of 20
- from 6 to 5 with a value of 12
- from 6 to 7 with a value of 11

If the nodes were cities and the network a map, the values could be distances. This network can be pictured like this:



The network may be represented by a matrix where rows represent source nodes and columns represent sink nodes. For example, the variable *SPM* below contains a 6 in row 2 column 3 because the directed arc from node 2 to node 3 is labeled with a 6:

```
      SPM
0  9  14  0   0   0   0
0  0   6  7  11   0   0
0  0   0  2   0  19   0
0  0   0  0  16   8   0
0  0   0  0   0   0  20
0  0   0  0  12   0  11
0  0   0  0   0   0   0
```

Note that in the matrix representation of the network, the sink node is represented by a row that is all zero because it is the node for which there is no next node. In order to keep the functions simple, a function *SETUP* is used to define some variables that describe the problem. Entering:

```
      SETUP SPM
```

establishes the following global variables that define the problem:

```
NODES - the list of node numbers
SIZE - the number of nodes
NETWORK - a copy of the network matrix
CM - a connection matrix (network matrix with 1 on each arc)
```

The function *PATHSFROM N* uses the global variables and computes all paths from the given node N to the sink node (number 7 in this example):

```
      PATHSFROM 4
 4 5 7   4 6 5 7   4 6 7
```

Since this network starts from node 1, you compute all paths as follows:

```
      ⍴PATH←PATHSFROM 1
14
```

Fourteen different paths are discovered. Here is the first path:

```
      ↑PATH
1 2 3 4 5 7
```

Here is the third path:

```
      3⊃PATH
1 2 3 4 6 7
```

The function *ARCS* computes the value on each arc of a path. If there are N nodes in a path, there will be N-1 arcs:

```
      ARCS ↑PATH
9 6 2 16 20
```

You can compute the arcs of all paths by using the **each** operator:

```
      A←ARCS¨PATH
```

The value of a path is the sum of its arc values. Here is the value of the first path:

```
      +/ARCS ↑PATH
53
```

The *VALUE* function combines the summation and the *ARCS* function:

```
      VALUE ↑PATH
53
```

The value of each path is easily computed:

```
      +/¨ARCS¨PATH
53 57 36 66 45 52 56 35 40 52 56 35 65 44
      VALUE¨PATH
53 57 36 66 45 52 56 35 40 52 56 35 65 44
```

The minimum of the sums is the shortest path through the network and the maximum of the sums is the maximum-length path through the network which is called the critical path in a PERT network:

```
      V←VALUE¨PATH
      ⌊/V
35
      ⌈/V
66
```

The following exhibits the paths which are shortest:

```
      (V=⌊/V)/PATH
 1 2 4 6 7  1 3 4 6 7
```

If the network contains a loop, the programs will not terminate. You stop a program that is running by pressing the break key (or sometimes the Esc key, Attn key, or Ctrl-Break). It would be instructive for you to modify *PATHSFROM* so it remembers which nodes it has visited and avoids loops. This is not straightforward.

The network examples in this section are adapted from the paper "APL2 and Basic Algorithms of OR," Norman Thomson, Proceedings of SEAS AM 1989, Amsterdam.

The IE workspace

The IE workspace contains an inferencing engine for a production rule based expert system. An inference engine is just a fancy name for a program that applies rules in an organized fashion. When an inferencing program applies rules to the known facts to produce new facts, the program is doing forward chaining. When an inferencing program applies rules to a desired conclusion in an attempt to prove that it is true, the program is doing backward chaining. The programs supplied in this workspace are designed to do forward chaining on rules like those in the following example.

Here are six production rules for animals:

1. If eyes face forward, teeth are sharp, it has claws, and it is class mammal, then it is a carnivore.
2. If eyes face forward, it has claws, and it is class bird, then it is a carnivore sometimes.
3. If phylum is chordata, birth is live, blood is warm, and skin is pliant, then class is mammal.
4. If phylum is chordata, birth is eggs, blood is warm and skin is feathers, then class is bird.
5. If skeleton is endo then phylum is chordata.
6. If skeleton is endo then cartilag.

In these rules, names such as "eyes" and "phylum" are treated as variables that could take different values. For example, in the first rule, the variable "eyes" has the value "forward." This can be represented in APL2 by a two-item vector as follows:

```
      'EYES' 'FORWARD'
```

Once you decide that *FORWARD* is the representation of the value "face forward," you must consistently use the same representation every place that the value is used. The programs do not know what the values mean.

The IF part of the first rule has four variables with values and can be represented by a four-item vector of pairs:

```
('EYES' 'FORWARD')('TEETH' 'SHARP')('CLAWS' 'HAS')('CLASS' 'MAMMAL')
```

This same structure is used for the representation of the IF parts of each of the six rules. The last two rules have only one IF part and so they must be represented by a one-item vector of pairs.

In the simple example above, each THEN part consists of a single variable and a single value. In general, there could be more than one variable value pair so the THEN parts must also be represented as a vector of pairs. In this example, the THEN parts would each be a one-item vector of pairs. A rule can be represented as a two-item vector where the first item is the IF part and the second item is the THEN part. The full set of rules could be represented as a six-item vector of IF-THEN pairs but in this workspace they are represented as a six by 2 matrix. Here is the DISPLAY of the rules for the animal example:

```
      DISPLAY RANIMAL
┌→──────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
↓ ┌→──────────────────────────────────────────────────────────────────────────┐ ┌→──────────────────┐            │
  │ ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐   │ │ ┌→──────────────┐ │ ┌→──────┐  │
  │ │ EYES │ FORWARD│ │ TEETH │ SHARP│ │ CLAWS │ HAS  │ │ CLASS │ MAMMAL│   │ │ │ CARNIVORE │ TRUE│ │ ┌→──┐ │  │
  │ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘   │ │ └~~~~~~~~~~~~~~┘ │ │ │ ⍬ │ │  │
  └~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~~~~~~┘ │ └~~~┘ │  │
  ┌→──────────────────────────────────────────────────────────┐ ┌→──────────────────┐            │ └~~~~~~┘  │
  │ ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐     │ │ ┌→──────────────┐ │            └~~~~~~~~~~~┘
  │ │ EYES │ FORWARD│ │ CLAWS │ HAS  │ │ CLASS │ BIRD  │     │ │ │ CARNIVORE │ SOMETIMES│ │
  │ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘     │ │ └~~~~~~~~~~~~~~┘ │
  └~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~~~~~~┘
  ┌→──────────────────────────────────────────────────────────────────────┐ ┌→──────────────┐
  │ ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐ │ │ ┌→──────────┐ │
  │ │ PHYLUM │ CHOR │ │ BIRTH │ LIVE │ │ BLOOD │ WARM  │ │ SKIN │ PLIANT │ │ │ │ CLASS │ MAMMAL│ │
  │ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ │ │ └~~~~~~~~~~┘ │
  └~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘
  ┌→──────────────────────────────────────────────────────────────────────┐ ┌→──────────────┐
  │ ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐ ┌→──────────────┐ │ │ ┌→──────────┐ │
  │ │ PHYLUM │ CHOR │ │ BIRTH │ EGGS │ │ BLOOD │ WARM  │ │ SKIN │ FEATHERS│ │ │ │ CLASS │ BIRD  │ │
  │ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘ │ │ └~~~~~~~~~~┘ │
  └~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~┘ └~~~~~~~~~~~~~~┘
  ┌→──────────────┐                                                        ┌→──────────────┐
  │ ┌→──────────┐ │                                                        │ ┌→──────────┐ │
  │ │ SKELETON │ ENDO│ │                                                  │ │ PHYLUM │ CHOR│ │
  │ └~~~~~~~~~~┘ │                                                        │ └~~~~~~~~~~┘ │
  └~~~~~~~~~~~~~~┘                                                        └~~~~~~~~~~~~~~┘
  ┌→──────────────┐                                                        ┌→──────────────┐
  │ ┌→──────────┐ │                                                        │ ┌→──────────┐ │
  │ │ SKELETON │ ENDO│ │                                                  │ │ CARTILAG │ YES│ │
  │ └~~~~~~~~~~┘ │                                                        │ └~~~~~~~~~~┘ │
  └~~~~~~~~~~~~~~┘                                                        └~~~~~~~~~~~~~~┘
```

There are two main functions in this workspace: *IECOMPILE* which analyzes the rule matrix and extracts information into global variables to be used for inferencing; and *IEFORWARD*1 which uses the information in the global variables to do forward chaining inferencing until no rules can be applied.

IECOMPILE

The most visible result of executing the *IECOMPILE* function is to prompt you for the values of variables used in the rules. If a value for a variable is known, respond with the value. If a value is not known, just press enter. Here is the execution of *IECOMPILE* for the animal example assuming that the values of the variables "carnivore" and "phylum" are not known:

```
      IECOMPILE RANIMAL
GIVE VALUES FOR VARIABLES

EYES
FORWARD

TEETH
SHARP

CLAWS
HAS

CLASS
BIRD

CARNIVORE


PHYLUM


BIRTH
LIVE

BLOOD
WARM

SKIN
PLIANT

SKELETON
ENDO

CARTILAG
NO
```

The program records the values of variables in the global variable *VARVAL* which in this case is an 11 by 2 matrix:

```
      ρVARVAL
11 2
      VARVAL
 EYES       FORWARD
 TEETH      SHARP
 CLAWS      HAS
 CLASS      BIRD
 CARNIVORE
 PHYLUM
 BIRTH      LIVE
 BLOOD      WARM
 SKIN       PLIANT
 SKELETON   ENDO
 CARTILAG   NO
```

The inferencing process will fill in the unknown values if it is possible, by applying the rules.

In addition to the values of variables, *IECOMPILE* builds the following globals:

- *NRULE* - the number of rules.
- *IFPARTS* - a vector of any variable value pair that occurs in an IF part of a rule.

- *VARIABLES* - a vector of the variables that are mentioned in rules.
- *VAR_IF* - a matrix of variable names versus IF parts - contains a 1 if a variable exists in the IF part.
- *IF_RULE* - a matrix that records which IF parts occur in which rules.
- *VAR_RULE* - a matrix that records which variables occur in which rules.

IEFORWARD1

*IEFORWARD*1 does forward chaining inferencing by applying rules until no more rules can be applied. It does this by first identifying rules that have all true IF parts. It applies all these rules on the first iteration. On subsequent iterations, the function deletes from the set of rules with true IF parts any rule that has not had a change of a variable value. Here is the execution of *IEFORWARD*1 showing that, after three iterations of applying rules, there are no more rules that can be applied:

```
      IEFORWARD1
 FIRE RULES  2 5 6
 FIRE RULES  3
 FIRE RULES  1
END OF INFERENCING
```

As a result of the inferencing, values have been given to variables that did not have values. In this example, all variables now have a value:

```
      VARVAL
 EYES       FORWARD
 TEETH      SHARP
 CLAWS      HAS
 CLASS      MAMMAL
 CARNIVORE  TRUE
 PHYLUM     CHOR
 BIRTH      LIVE
 BLOOD      WARM
 SKIN       PLIANT
 SKELETON   ENDO
 CARTILAG   YES
```

These functions represent a very simple inferencing technique. More sophisticated techniques can be used to reduce the actual amount of computing done. The papers referenced below contain discussions of various criteria for choosing the rules to be applied.

The inferencing examples in this section are based on the papers:

"A Boolean Array Based Algorithm in APL for Forward Chaining in Rule Based Production Expert Systems," Fordyce, K. and Sullivan, G. 1987, *APL Quote Quad*, APL87 Conference Proceedings, Vol. 16, No. 3.

"BABIE: Boolean Array Based Inference Engines," Fordyce, K., and Sullivan, G. 1988, Proceedings of the 1987 APL Techniques in Expert Systems Conference, Syracuse University.

The IDIOMS workspace

An APL2 idiom is a short expression that does some interesting function. The symbolic nature of APL2 lets you recognize common expressions easily. Here is an example of the idiom that counts the number of leading blanks in a character string:

```
+/∧\STRING=' '
```

The idioms workspace is a full screen application that lets you search for idioms. Load the workspace and execute the main function *IDIOMS*. Follow the directions to see what is available. One interesting thing to try is to press PF6 to get a listing of the group headings for idioms. PF7 and PF8 can be used to scroll. This is a large workspace and can get *WS FULL* if there is not enough space (e.g. TSRs running or less than 640K memory).

The *IDIOMS* workspace was donated by Stan Cason, IBM Endicott.

Appendix A. Discovering APL2 -- Working With Arithmetic

This discovery exercise shows you how to do arithmetic functions that perform calculations on numbers. When you are finished you will have a little of your very own reference manual.

Here are some definitions that might be helpful as you do the tables.

argument data which is input to the function.

scalar A single number or a single character.

vector A list of single numbers (separated by a space) or single characters (called a *character string*).

function operation that applies to data and produces new data.

scalar function function that applies independently to each simple scalar in a uniform way.

monadic function function applied to only a right argument.

dyadic function function applied to both left and right arguments.

Here is some information about the tables in general:

- The first column gives you the name of the function, the syntax for the function using *L* for the left argument and *R* for right argument.
- The second column is for the result; that is, what the function does (fill in using your own words).
- The third column gives a sample problem (fill in the solution).

Here is some information about the representation and structure of data in the tables:

- The left and right arguments are numeric scalars or vectors.
- The arguments *conform* (have the same number of items).
- The results conform to the arguments.
- The function applies between corresponding items of conforming vectors.

The first table shows you functions that perform simple calculations. The first entry is done for you as an example.

Function and Syntax	Result	Example
ADD *L+R*	Sum of *L* and *R*	`1 2 3 + 4 5 6` `5 7 9`
DIVIDE *L÷R*		`8 ÷ 0.4`
MULTIPLY *L×R*		`13 × 34`

Function and Syntax	Result	Example
NEGATIVE *-R*		- ¯5 1
POWER *L*R*		2 * 4
RECIPROCAL *÷R*		÷ 5
RESIDUE *L|R*		8 | 9 15 24 32 44
SUBTRACT *L-R*		¯3 - 6

This second table looks at functions that do not actually do calculations but rather compare numbers, giving numeric results.

Function and Syntax	Result	Example
CEILING ⌈*R*		⌈ 3.4 ¯.3 ¯3.4
FLOOR ⌊*R*		⌊ 3.4 ¯.3 ¯3.4
MAGNITUDE |*R*		|6 ¯5 ¯4
MAXIMUM *L⌈R*		4 ⌈ 1
MINIMUM *L⌊R*		6 ⌊ 1

Appendix B. Discovering APL2 -- Working With Vectors

This discovery exercise shows you how to measure, create, manipulate, and select particular items from data arguments.

Here is some information about the tables in general:

- The first column gives you the name of the function and the syntax for that function.
- The second column is for the result; that is, what the function does (fill in using your own words).
- The third column gives a sample problem (fill in the solution).

Here is some information about the representation and structure of the data in the tables:

- The data used are both numeric and character.

  You should understand the following about character data:

  1. A simple character is a symbol surrounded by quotes.
  2. A list of characters (a vector) may be represented as a series of single characters (example: `'A' 'B' 'C'`) or as a character string (example: `'ABC'`).
  3. A blank space is a character when written inside quotes (example: `' '`).
- The arguments and the results do not conform (which makes them non-scalar functions).

This first table shows how to measure, create, and manipulate data.

Function and Syntax	Result	Example
SHAPE `⍴R`		`⍴ 1 2 3 4 5`
SHAPE `⍴R`	same as above	`⍴ 'LEMONS'`
INTERVAL `⍳R`		`⍳6`
CATENATE `L,R`		`'HI','HO',' ','HI'`

This next table will show you how to select certain items from a vector.

Function and Syntax	Result	Example
DROP *L↓R*	for a positive integer scalar L,...	`2 ↓ 'LEMONS'`
DROP *L↓R*	for a negative integer scalar L,...	`¯2 ↓ 'LEMONS'`
TAKE *L↑R*	for a positive integer scalar L,...	`2 ↑ 'LEMONS'`
TAKE *L↑R*	for a negative integer scalar L,...	`¯2 ↑ 'LEMONS'`
PICK *L⊃R*	for an integer scalar for L,...	`3 ⊃ 'LEMONS'`
FIRST *↑R*		`↑'LEMONS'`

Appendix C. Discovering APL2 -- Working With Arrays

This discovery exercise shows you how to create, measure, manipulate, and perform some simple arithmetic calculations using matrices as arguments. The functions used should be familiar to you as you saw them in the previous discoveries.

Here is a definition that might be helpful as you do this exercise:

array rectangular collection of zero or more items arranged along zero or more axes (directions). Each item in the array is single number, character, or another array.

matrix an array with two dimensions.

Here is some information about the tables in general:

- The first column gives you the name of the function and the syntax for that function.
- The second column is for the result; that is, what the function does (fill in using your own words).
- The third column gives a sample problem (fill in the solution).

Here is some information about the representation and structure of the data in this first table:

- The arguments are numeric and character scalars, vectors, or matrices.

  You should understand the following about matrices:

  1. Every row has the same number of items.
  2. Every column has the same number of items.
- The arguments and the results do not conform (which makes them non-scalar functions).

This first table shows how to measure, create, and manipulate data.

Function and Syntax	Result	Example
SHAPE ⍴*R*		⍴(2 3 ⍴ '*ABCDEF*')
RESHAPE *L*⍴*R*		2 3 ⍴ 1 2 3
RESHAPE *L*⍴*R*	same as above	2 3 ⍴ '*ABCDEF*'
RESHAPE *L*⍴*R*	same as above	3 4 ⍴ ⍳12

Function and Syntax	Result	Example
CATENATE *L*,*R*		12 , 2 3 ⍴ ⍳6

Here is some information about the representation and structure of the data in this next table:

1. The right argument used is a simple scalar number.
2. The left argument used is a matrix.
3. The result is a matrix of the same shape as the original matrix.

This next table will show you a few arithmetic operations using matrices.

Function and Syntax	Result	Example
ADD *L*+*R*		(2 3 ⍴ ⍳6) + 10
SUBTRACT *L*-*R*		(2 3 ⍴ ⍳6) - 10
MULTIPLY *L*×*R*		(2 3 ⍴ ⍳6) × 10
DIVIDE *L*÷*R*		(2 3 ⍴ ⍳6) ÷ 10

Appendix D. Discovering APL2 -- Working With Comparisons

This discovery exercise shows you how to do various comparisons with relational functions and Boolean functions. When you are finished you will have a little more of your very own reference manual.

Here is a definition that might be helpful as you do the first table:

Boolean vector a vector of 1s (TRUE) and 0s (FALSE).

Here is some information about the tables in general:

- The first column gives you the name of the function and the syntax for that function.
- The second column is for the result; that is, what the function does (fill in using your own words).
- The third column gives a sample problem (fill in the solution).

Here is some information about the data representation and structure in the first table:

- The arguments are conforming numeric vectors.
- The results are Boolean vectors conforming to the arguments.

This first table deals with relational functions.

Function and Syntax	**Result**	**Example**
EQUAL *L=R*		20 30 40 = 40 30 20
GREATER THAN *L>R*		20 30 40 > 40 30 20
GREATER THAN OR EQUAL TO *L≥R*		20 30 40 ≥ 40 30 20
LESS THAN *L<R*		20 30 40 < 40 30 20
LESS THAN OR EQUAL TO *L≤R*		20 30 40 ≤ 40 30 20
NOT EQUAL *L≠R*		20 30 40 ≠ 40 30 20

This second table deals with Boolean Functions. Here is a definition that might be helpful:

Boolean function a scalar function which takes Boolean arguments and returns Boolean results.

Here is some information about the data representation and structure:

- The arguments are Boolean vectors.

  The functions (scalar) apply between conforming vectors giving all four possible combinations:

  - 0 with 0 (FALSE-FALSE)
  - 0 with 1 (FALSE-TRUE)
  - 1 with 0 (TRUE-FALSE)
  - 1 with 1 (TRUE-TRUE)
- The results are conforming Boolean vectors (of the same length as the arguments).

Function and Syntax	**Result**	**Example**
AND *L∧R*		0 0 1 1 ∧ 0 1 0 1
OR *L∨R*		0 0 1 1 ∨ 0 1 0 1
NAND *L⍲R*		0 0 1 1 ⍲ 0 1 0 1
NOR *L⍱R*		0 0 1 1 ⍱ 0 1 0 1
NOT *~R*		~1 0

Appendix E. Discovering APL2 -- Working With Functions and Data

This discovery exercise will show you how to control functions with operators. These operators create whole families of new functions. When you are finished you will have a little more of your very own reference manual.

Here are some definitions that might be helpful as you do the first table:

operator operation that applies to functions and/or data and produces a new function as its result (called a derived function).

operand data or a function given to an operator as input.

derived function function formed by the application of an operator to its operand(s).

Here is some information about the table(s) in general:

- The first column gives you the name of the operator and the syntax using *LO* for the left operand and *RO* for the right operand.
- The second column is for the result; that is, what the derived function does (fill in using your own words).
- The third column gives a sample problem with an operator and a selected operand (fill in the solution).

Here is some information about the derived functions used in the first table:

- The operator used is the *slash* (/) symbol.
- Different system-defined functions are applied with numeric vectors.
- The results are numeric scalars.

As you try to describe the results in this first table, remember the right-to-left rule of APL2.

Operator and Syntax	Result	Example
REDUCE *LO/ R* .	Using the addition function as an operand,...	`+/ 1 2 3 4 5`
REDUCE *LO/ R*	Using the subtraction function as an operand,...	`-/ 1 2 3 4 5`
REDUCE *LO/ R*	Using the multiplication function as an operand,...	`×/ 1 2 3 4 5`
REDUCE *LO/ R*	Using the division function as an operand,...	`÷/ 1 2 3 4 5`

Here is some information about the derived functions in this second table:

- The operator used is *Array Product* (`.`).
- Both the left and the right operand are applied. The right operand is a function. The left operand is replaced by the "jot" (`∘`) symbol which is a place holder.
- Both the left and right arguments are applied. They are numeric vectors.
- The results are matrices.

Operator and Syntax	**Result**	**Example**
OUTER PRODUCT *L ∘.RO R*		`10 20 30 ∘.+ ⍳5`
OUTER PRODUCT *L ∘.RO R*		`10 20 30 ∘.- ⍳5`
OUTER PRODUCT *L ∘.RO R*		`10 20 30 ∘.× ⍳5`
OUTER PRODUCT *L ∘.RO R*		`10 20 30 ∘.÷ ⍳5`

Appendix F. APL Expressions for Some Mathematical Algorithms

This section contains some short APL2 expressions that do useful things.

Round Number(s) N to D decimal places

```
(10*-D)×⌊.5+N×10*D
```

```
N: numeric scalar, vector, or array
D: integer (if negative, N is rounded to nearest 10*-D)
```

Average of a vector V

```
(+/V)÷⍴V
```

V: numeric vector

The first N even integers

```
2×⍳N
```

N: positive integer

The first N odd integers

```
¯1+2×⍳N
```

N: positive integer

Multiplication table of size M

```
(⍳M)∘.×⍳M
```

M: positive integer

Factors of a number N (including N itself)

```
(0=(⍳N)|N)/⍳N
```

N: integer

Factors of a number N (excluding N)

```
(0=(⍳⌊.5×N)|N)/⍳⌊.5×N
```

N: integer (Note that this expression uses the fact that no factor of N can exceed N÷2.

The expressions in this section are adapted from the book "APL Programs for the Mathematics Classroom" by Norman D. Thomson, copyright 1989, Springer-Verlag, ISBN 0-387-97002-9

Appendix G. References/Support

This section discusses national codepages, tells you how to build a TryAPL2 workspace from an APL2/PC workspace, and gives you sources of documentation.

National codepages

If you are using a national codepage, you should enter TryAPL2 with the command:

```
tryapl2n
```

You may need to customize tryapl2n.bat. The bat file contains remarks saying how you might want to change it.

Your CONFIG.SYS and AUTOEXEC.BAT will normally already be modified to allow this support. Here are examples of modifications that work in Germany with an EGA:

CONFIG.SYS

```
install=c:\dos\nlsfunc.exe c:\dos\country.sys
country = 049,437,\dos\country.sys
```

AUTOEXEC.BAT

```
mode con cp prep=((437) c:\dos\ega.cpi)
mode con cp select=437
keyb gr,437,\dos\keyboard.sys
```

Support Information for Teachers

You may make copies of the TryAPL2 disk and give them to students. You must supply all the files, not a subset. You may add workspaces to TryAPL2 to provide additional demonstrations of APL programming style or in support of use of TryAPL2 for classroom teaching. If you use the purchased APL2/PC product to prepare a workspace for use with TryAPL2, use the *TRYSAVE* program reproduced below. A TryAPL2 workspace is an AP211 file with a file extension of "TRY." Write variables to the file by prefixing a V to the name of the variable. Write defined functions by writing the $\square CR$ with an F prefixed to the name of the function. Write defined operators by writing the $\square CR$ with an O prefixed to the name of the operator. See the documentation that comes with APL2/PC for details on AP211. Here is a sample function to write all objects in a workspace to a AP211 file:

```
      ∇TRYSAVE Δ*FN*;Δ211;Δ*T*;Δ*LIST*;Δ*RC*;Δ*VAL*;Δ*TS*
[1]   ⍝ WRITE TryAPL2 WORKSPACE
[2]    Δ*TS*←⎕*TS*
[3]    Δ*LIST*←((⊂[2]⎕*NL* 2 3 4)~¨' ')~'*TRYSAVE*' '*TRYLOAD*'
[4]    Δ*LIST*←('Δ'≠↑¨Δ*LIST*)/Δ*LIST*←Δ*LIST*,⊂'⎕*LX*'
[5]    Δ*T*←211 ⎕*SVO* 'Δ211'
[6]    Δ*T*←1 ⎕*SVC* 'Δ211'
[7]   Δ*L*0:Δ211←'*CREATE* '(Δ*FN*,'.*TRY*')512
[8]    →(0≠↑Δ*RC*←Δ211)/Δ*ER*1
[9]    Δ211←'*USE* '(Δ*FN*,'.*TRY*')
[10]   →(0≠↑Δ*RC*←Δ211)/Δ*ER*2
[11]  Δ*L*1:→(0=⍴Δ*LIST*)/Δ*END*
[12]   →(2 3 4=⎕*NC*↑Δ*LIST*)/Δ*VAR* Δ*F* Δ*OP*
[13]  Δ*NXT*:Δ*LIST*←1↓Δ*LIST*
[14]   →Δ*L*1
[15]  Δ*VAR*:Δ211←'*SET*'('*V*',↑Δ*LIST*)(⍎↑Δ*LIST*)
[16]   →(0≠↑Δ*RC*←Δ211)/Δ*ER*3
[17]   →Δ*NXT*
[18]  Δ*F*:Δ211←'*SET*'('*F*',↑Δ*LIST*)(⎕*CR*↑Δ*LIST*)
[19]   →(0≠↑Δ*RC*←Δ211)/Δ*ER*3
[20]   →Δ*NXT*
[21]  Δ*OP*:Δ211←'*SET*'('*O*',↑Δ*LIST*)(⎕*CR*↑Δ*LIST*)
[22]   →(0≠⍴Δ*RC*←Δ211)/Δ*ER*3
[23]   →Δ*NXT*
[24]  Δ*END*:Δ211←'*SET*' '?Δ*TS*' ⎕*TS*
[25]   Δ211←'*RELEASE*'
[26]   →(0≠↑Δ*RC*←Δ211)/'∘'
[27]   →0
[28]  Δ*ER*1:Δ211←'*DROP*'(Δ*FN*,'.*TRY*')
[29]   Δ211←'*RELEASE*'(Δ*FN*,'.*TRY*')
[30]   →Δ*L*0
[31]  Δ*ER*2:'*USE FAILED* ' Δ*RC*
[32]   →0
[33]  Δ*ER*3:'*SET FAILED*' Δ*RC*(↑Δ*LIST*)
```

Here is a sample function that will read a TryAPL2 211 file into an APL2/PC workspace:

```
      ∇TRYLOAD Δ*FN*;Δ211;Δ*T*;Δ*LIST*;Δ*RC*;Δ*VAL*
[1]   ⍝ READ TryAPL2 WORKSPACE
[2]    Δ*T*←211 ⎕*SVO* 'Δ211'
[3]    Δ211←'*USE*'(Δ*FN*,'.*TRY*')
[4]    →(0≠↑Δ*RC*←Δ211)/Δ*ER*1
[5]    Δ211←'*LIST*' '*NAMES*'
[6]    Δ*LIST*←Δ211
[7]   Δ*L*1:→('*VFO?*'=↑Δ*LIST*)/Δ*VAR* Δ*FO* Δ*FO* Δ*NXT*
[8]    '*ILLEGAL OBJECT*'(Δ*LIST*[⎕*IO*;])
[9]   Δ*NXT*:Δ*LIST*←1↓[⎕*IO*]Δ*LIST*
[10]   →(0=↑⍴Δ*LIST*)↓Δ*L*1
[11]   →0
[12]  Δ*VAR*:Δ211←'*GET*'(Δ*LIST*[1;]~' ')
[13]   (Δ*RC* Δ*VAL*)←Δ211
[14]   ⍎(1↓Δ*LIST*[⎕*IO*;]),'←Δ*VAL*'
[15]   →Δ*NXT*
[16]  Δ*FO*:Δ211←'*GET*'(Δ*LIST*[1;]~' ')
[17]   (Δ*RC* Δ*VAL*)←Δ211
[18]   Δ*T*←⎕*FX* Δ*VAL*
[19]   →Δ*NXT*
[20]  Δ*ER*1:'*USE FAILED* ' Δ*RC*
```

APL2 Interfaces

APL2 systems come with sophisticated interfaces to graphics, databases, dialog managers, networks, files, and programs written in other languages. Most of these interfaces are not supplied with TryAPL2.

Differences from the full APL2 product

TryAPL2 is the full APL2 language provided by the purchased product. Some parts of the environment are simulated and as such may exhibit slightly different behavior from the purchased product.

- Errors do not cause suspension. When an error occurs, TryAPL2 generates a right arrow → to clear the stack. In the purchased product, errors cause suspension of programs at the point of the error. This allows you to investigate the cause of an error, fix it, and continue execution from where you left off.
- The del-editor is simulated and is not the same as the purchased product in every respect.
- You cannot edit a line of a function wider than the screen.
- *S*Δ is not supported because suspension is not allowed.
- The display from)*OFF* is different.
- The display from)*LIB* contains less information and is formatted differently. Library numbers are not supported.
-)*SAVE* and)*OUT* both write AP211 files rather than APL2 workspaces (.APL files) and transfer files (.ATF files).
- Names can be up to 17 characters in length. Longer names will cause a failure on)*SAVE* and)*OUT*.
-)*LOAD*,)*IN*, and)*COPY* read AP211 files rather than access APL2 workspaces.
-)*OUT* does not accept a name list.
-)*PCOPY* and)*PIN* are not supported.
-)*NMS* etc. do not accept specification of ranges.
- Many auxiliary processors are not provided.
- ⍛ is not supported.
- Locked functions cannot be saved or loaded.
- □*LC* contains extraneous line numbers.
- Complex arithmetic is not supported.
- The 32-bit interpreter that allows use of memories larger than 640K is not supported.
-)*LOAD* of a workspace that doesn't exist leaves a clear workspace.

APL2 Publications you can purchase

- APL2 at a Glance, Brown, Pakin, and Polivka, Prentice-Hall, Englewood Cliffs, New Jersey, ISBN 0-13-038670-7, 1988, IBM number SC26-4676.
- APL2 Programming: Language Reference, Version 2, SH21-1061, IBM Corporation.
- APL2 Programming: Language Reference, Version 1, SH20-9227, IBM Corporation.

- APL2 - ein erster Einblick, Brown, Pakin, and Polivka, Springer Verlag, Berlin, New York, ISBN 3-540-51611-5, 1989, translated by Heinz-Albert Badior.
- APL Programs for the Mathematics Classroom, Norman D. Thomson, Springer Verlag, Berlin, New York, ISBN 0-387-97002-9, 1989

APL2 Publications from IBM for free

There is no charge for these reports and you may make additional copies without permission so long as the entire report is copied without modification. They may be ordered from:

APL Products
IBM Santa Teresa, Dept. M46/D12T
P.O. Box 49023
San Jose, Calif. 95161-9023

Table 1. Publications for free from IBM Santa Teresa

Number	Title	Author
TR 03.247	The Principles of APL2	J.A. Brown
TR 03.265	Graphics Applications Using Complex Numbers in APL2	J.A. Brown H. P. Crowder
TR 03.266	Migrating Applications to APL2	M.T. Wheatley
TR 03.267	APL2: Exploiting DB2 and SQL/DS	J.A. Brown H. P. Crowder
TR 03.274	Multi-User SQL Applications in APL2	J.A. Brown
TR 03.281	Algorithms for Artificial Intelligence in APL2	J.A. Brown E. Eusebi L. Groner J. Cook
TR 03.286	The APL2 Name Association Facility: Understanding the APL-FORTRAN Connection	H.P. Crowder
TR 03.288	Interactive Programming in a Multifaceted Environment	H.P. Crowder D. Dunbar
ACM 554891	APL2 and SQL: A Tutorial	Nancy Wheeler
TR 01.A845	APL2 Phrases	Stan Cason

The following Technical Reports may be ordered from other IBM locations:

Table 2. Publications for free from other IBM locations		
Number	**Title**	**Author**
RC 11025	A computer Gallery of Mathematical Physics - a Course Outline	G.J. Chaitin IBM Yorktown Heights Route 134 Yorktown Heights, NY
TR 21.1078	APL2 Version 1 Release 3 Primitive Function Performance on the IBM 3090 Vector Facility	M. Van Der Meulen M. Morreale IBM Kingston Neighborhood Road Kingston, NY 12401
TR 21.1292	Primitive Function performance of APL2 Version 1 Release 3 (with SPE PL34409) on the IBM 3090/S Vector Facility	M. Van Der Meulen M. Morreale IBM Kingston Neighborhood Road Kingston, NY 12401

Key stickers

Keyboard Decals are available from your IBM representative as order number SC33-0604-0.

Keytops

Keytops are available from your IBM representative as follows:

Description	Manufacturing Part Number	Mechanicsburg Form Number
APL2 Keycaps, US/UK	1395624	SX80-0270
APL2 Keycaps, German upgrade	1397152	SX23-0452
APL2 Keycaps, French upgrade	1397153	SX23-0453
APL2 Keycaps, Italian upgrade	1397154	SX23-0454

Part Number 1395624 is needed to complete each of the other sets; Part Numbers 1397152 through 1397154 are supplemental to the US/UK set.

These keycaps may be used on the following keyboards:

IBM PS/2 (except as noted)
IBM RISC System/6000

They are NOT suitable for the following keyboards:

* IBM PS/1
* IBM PS/2 Model P70
* IBM PS/2 Model P75

They will physically fit on a PS/2 Model 25 with the "space-saving keyboard," but they lack some of the markings of that original keyboard.

How to order the full product

In North America, the PC or PS/2 product is ordered by calling the toll free number 1 (800) IBM-CALL and asking for RPQ# RJ0411. You can also call IBM DIRECT at the toll-free number (800) IBM-2468 and asking for part number 6242936. Alternatively, contact your IBM representative and ask for 5799-PGG. The product costs 500 US Dollars and volume discounts are available. On an 80386 or 80486 machine, workspaces up to 15 megabytes are supported. An 80387 is required with an 80386 to use large workspaces. Be sure to send in your registration card.

In Europe, the product is ordered by contacting your authorized PC or PS/2 dealer and asking for product number 5604-260 or part number 38F1753.

APL2/6000 may be ordered by contacting your IBM representative and asking for program number 5765-012.

APL2/370 Version 2 may be ordered by contacting your IBM representative and asking for program number 5688-228. APL2 Application Environment is a run time version of APL2/370 on which APL2 applications can be run but not developed. It may be ordered from your IBM representative by asking for program number 5688-229.

Acknowledgements

I'd like to thank the many people who contributed suggestions, content, and support. These include Doug Aiton, Manuel Alfonseca, Albert Badior, Tom Bundros, Stan Cason, Dick Conner, Paul Conrad, Coke Costello, Rich Cunningham, Barry Dorfman, Dick Dunbar, Ted Edwards, Ken Fordyce, Lyle Gayne, Jacques Gourdon, Alan Graham, Deen Hamid, Brent Hawks, Jim Henry, Tom Jacobs, Evan Jennings, Curtis Jones, Erik Kane, Mike Kingston, Bernard Landaud, Dieter Lattermann, Guido Leeten, David Liebtag, Gary Logan, Jon McGrew, John McPherson, Mike Van Der Meulen, John Mizel, Peggy Millar, Mario Morreale, Dick Oates, Tom O'Brien, Bruce Pabst, Howard Richmond, Paula Schineller, David Selby, Dick Stitt, Norm Thomson, Ray Trimble, Karl Vincena, Mike Wheatley, Nancy Wheeler, Dave White, and Ron Wilks.

Jim Brown
APL2 Products and Service
August 1991

Change History

- Version 1.00 - August, 1989
- Version 1.01 - September, 1989. New "del" editor, *END* variable in *SEARCH* workspace, corrected *IDIOMS* workspace.
- Version 1.02 - October, 1989. Fixed time stamp on *LOAD* and *SAVE*, fixed "min" and "max" in *STATS* workspace. New *IDIOMS* workspace. □*LX* supported and used in supplied workspaces.
- Version 1.03 - November, 1989. More reliable checking for presence of a math coprocessor.
- Version 1.04 - February, 1990. National codepage support.
- Version 1.05 - October, 1990. High School Mathematics Class sections added, *STATS* workspace and documentation simplified.

- Version 2.00 - October, 1991. Documentation on the disk, AP207 added, AP206 deleted, GRAPHS, KEYS, TRYDOC, CLEANSPACE workspaces.

Keyboard templates

Keyboard templates are supplied in the companion document TRYTEMPL.HPC. The file is formatted to print on LaserJet printers.

Diskette labels

If you make copies of the TryAPL2 diskettes, please paste these labels onto your copies. Cut out the rectangular area bounded by the inside corners of the small squares. The small boxes in the corners are not part of the label.