CAPLIB^2: APL for Programmers in C

Thomas Glen Smith

May 9, 1996

Address inquiries to

Thomas Glen Smith
3154 West Shady Lane
Neenah, WI 54956

Contents

Chapter 1

Beginning

1.1 Trademarks

APL2 and IBM are trademarks of IBM Corporation. CAPLIB^2 is my trademark.

1.2 Introduction

APL was invented by Dr. Ken Iverson while he worked at Harvard. He published a book, *A Programming Language*, in 1962, and continued his work on the language at IBM. IBM marketed an APL Interpreter, documented in IBM publication GC26-3847-4, **"APL Language"**, Fifth Edition, July 1978. In 1992, I published CAPLIB, an APL interpreter, written in C, that was an implementation of the APL language as documented in IBM's publication. That same year, IBM announced APL2, a new and improved APL. I sold three copies of CAPLIB.

Only slightly daunted, I began work on CAPLIB^2, an implementation of APL2 as documented in IBM publication SH21-1061-00, **"APL2 Programming: Language Reference"**, First Edition, March 1992.

About the Name

Everything needs a name, especially a product for public use. This one contains a **LIB**rary of routines you can call from your **C** program to do **APL** operations, including extensions in IBM's APL**2**, so I called it **CAPLIB^2**. It's not a perfect acronym, like **CO**mmon **B**usiness **O**riented **L**anguage, or **A** **P**rogramming **L**anguage. But it's at least pronounceable, and it's mine.

CAPLIB^2 is Free

Yup, it's (gulp) free. CAPLIB was proprietary, though I'd sell you a copy of the source. CAPLIB^2 is free software, protected under the GNU General Public License, a complete copy of which is in Appendix Q on page 471. Briefly, this means you can give away or even sell copies of CAPLIB^2, so long as you conspicuously attach a copy of the GNU copyright notice and disclaimer of warranty, **and supply copies of the source, and permit your customers to give away free copies of what you sold or gave them**.

You can also modify CAPLIB^2 source, or add to it, so long as you clearly indicate what you've changed or added (so the next person who uses it won't blame me for your mistakes), **and protect your changes or additions under the GNU General Public License**. In other words, your stuff must be free, too. In fact, if it's good, I want a copy!

Why would I spend fifteen years of my life developing CAPLIB^2, only to give it away? Well, it's not out of altruism. I like making a living as an independent computer programmer, so if putting the source for CAPLIB^2 in a bank vault and selling only copies of the executables would work, I'd do it. But I don't think that'll do much longer, even for the big companies like IBM and Microsoft.

Reading an article by Kevin Kelly titled "New Rules for the New Economy," in the September 1997 issue of Wired magazine finally convinced me to make CAPLIB^2 free. Though he describes the "Network Economy," the Internet being it's principal manifestation, he's equally describing the success or failure of computer programs. He says,

"There is no future for hermetically sealed closed systems in the Network Economy. The more dimensions accessible to member input and creation, the more increasing returns can animate the network, the more the system will feed on itself and prosper. The less it allows these, the more it will be bypassed. ... The Network Economy rewards schemes that allow decentralized creation and punishes those that don't."

Like the dinosaurs, the day of proprietary, OCO (Object Code Only, a term originated I think by IBM) software is over. On the Internet, free software can adapt and mutate, quickly getting better, while propriety stuff can't, because changes only come with the next release from the vendor. Free programs behave like entities in a biome, developing through a process of natural selection.

I still plan on making money from CAPLIB2 by offering a service. I'll be examining the "mutations," incorporating good ones that're downward compatible in a new version. The source for each new "mutated" CAPLIB2 will still be free, as it must to comply with the GNU General Public License, but you'll pay a service charge to get a copy from me. Of course, others may also offer such a service. It's up to me to make it worth your while to get yours from me.

What is CAPLIB2

CAPLIB2 contains an interpreter that lets you program directly in APL on a standard IBM PC or compatible, presently under either DOS or the Linux operating system. Since it's written in C, it can be easily ported to other operating systems. In fact, although it took me years to complete the initial version for DOS, it required less than two weeks to then get it working under Linux.

With CAPLIB2 you can interactively explore problem solutions in APL. Once you've found the best solution, if you need greater efficiency, you can incorporate that logic in a C program through calls to routines in the CAPLIB2 C-library. There are over seven hundred routines there, compared to four hundred in CAPLIB.

Almost any APL statement involving standard APL functions and operators can be directly translated to a single C language statement using calls to CAPLIB2 routines. Here's a brief illustration of how an APL statement can be converted: **Inner product** is a standard APL operator, discussed in detail on page 193. If **LEFT** and **RIGHT** are its two arguments, the shape of its result is given by the APL2 expression -

$$(^{-}1 \downarrow \rho\, LEFT), 1 \downarrow \rho\, RIGHT$$

The equivalent C programming expression, copied verbatim from the CAPLIB2 routine that implements **inner product**, is listed below:

```
dimcb = perm(
     cat(
          drop(
               iscalar(-1),
               shape(left)
          ),
          drop(
               iscalar(1),
               shape(rite)
```

```
            ),
            indxorg
        )
    );
```

Not everything documented in **"APL2 Programming: Language Reference"** is implemented in CAPLIB2, but most of it is. This book will define what's different, and what's omitted.

1.3 Interactive APL

Starting and Stopping the APL Interpreter

Assuming either you're in the directory containing the **apl2** binary, or it's in your DOS or Linux shell *search path*, key "**apl2**" and press **Enter** to interact directly with the CAPLIB2 APL interpreter. It'll prompt you with "**apl>**". To stop the interpreter and return to a DOS or Linux shell prompt, key "**)off**" and press **Enter**.

APL Symbol Set

If you've used APL before and you're reading this for the first time, you may think "**)off**" is a typographical error. It should read ")OFF" since the APL symbol set doesn't include lower case letters. Figures 1.1 and 1.2 on pages 5 and 5 respectively show the layouts of typical APL and standard PC keyboards. Obviously some sort of mapping needs to take place in order to use APL on a standard PC.

CAPLIB2 maps the upper case letters in the APL symbol set to the lower case letters of the standard PC keyboard. Mapping for the other symbols can be seen by comparing Figures B.1 and B.2 on pages 350 and 351 in Appendix B. These tables respectively show the APL symbol set and the corresponding symbols used in CAPLIB2. Appendix B gives other details such as the ASCII codes used for each symbol.

Interactive Help

The same distribution media containing the the CAPLIB2 APL2 interactive interpreter also contains an APL workspace called **help.apl**. The relative path is **cap2/help/help.apl**. While using APL2 interactively, assuming the current directory contains **help.apl**, you can copy help information from it into the active workspace in main storage with the command -

)copy help helpgroup

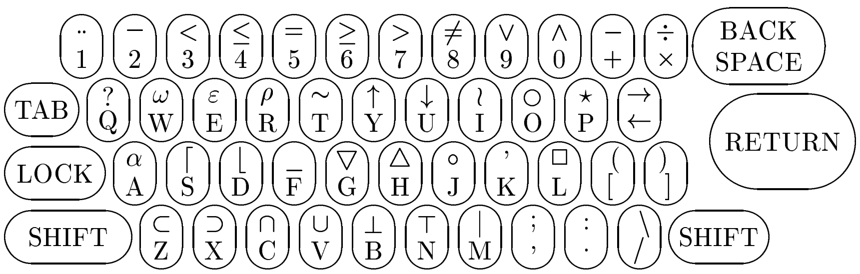


Figure 1.1: Typical APL Keyboard

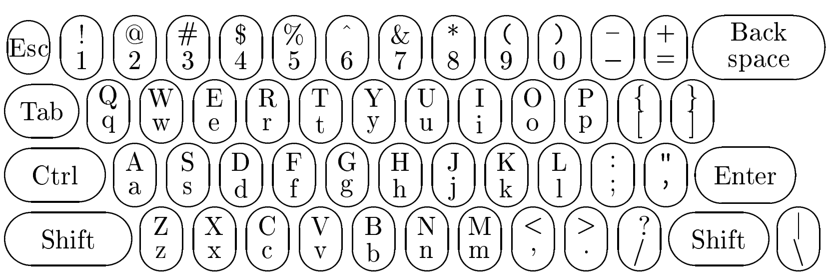


Figure 1.2: Typical PC Keyboard

This copies a user function named **help** and two APL variables, **helpmsg** and **helptab**, into the active workspace.

The **help** user function accepts a single argument string, which may be one of three things:

- A keyboard character, "`help 'R'`" for example.
- The name of some APL function, "`help 'reshape'`" or `help 'index generator'`."
- The name of an APL symbol, such as "`help 'rho'`."

In the case of a keyboard character, **help** tells you what functions are invoked in an APL statement when that particular character is encountered. If you give it the name of an APL symbol, it'll tell you what functions are invoked by that symbol. Given the name of an APL function, it'll tell you what keyboard characters invoke that function, and will also give an example using it.

Following is a dialogue showing the kinds of information the **help** function can provide:

```
apl> help 'rho'

        R   rho symbol, reshape, shape

apl> help 'R'

        R   rho symbol, reshape, shape
        R   reshape, 2 2RI4 is 2-by-2 matrix of 1 2 3 4
        R   shape, R2 2RI4 is 2 2
```

1.4 Tutorial

APL Versus Other Programming Languages

Most of us who claim to be programmers were introduced to the craft with some procedure-oriented language such as BASIC, FORTRAN, PL/1, or C. The keywords and syntax may differ from one to the next, but a programmer uses similar logic with each language. Operators typically deal only with one item at a time, and you must define a loop to iteratively process arrays of items. Here's a C program that adds 5 numbers and prints the sum:

```
%sam138
#define INCLUDES APLMAIN+STDIO
#include "includes.h"
main()
{
     static double a[5] = {11.0, 22.1, 3.2, 43.5, 71.4};
     double sum=0.0;
     int i;

for ( i=0; i<5; i++ ) sum += a[i];
     printf("sum = %5.2f",sum);
}

sum = 151.20
```

Contrast the above C program with the solution of the same problem in APL:

```
apl> + / 22.1 3.2 43.5 71.4

    151.2
```

First of course it's concise, and there are pros and cons to this. There's the advantage of a smaller source to maintain, but in a more complicated program, a few months after it was written it can be difficult to understand the original intent. You've heard of read-only memory? Some critics of APL refer to it as the "write-only" language.

A major advantage in APL is that structure information is carried along with data. If you wanted to change the problem to find the sum of six numbers instead of five, you'd need to change the C program **for** statement to test for **i** less than 6 instead of 5. But in APL all you change is the data:

```
apl> + / 22.1 3.2 43.5 71.4 89.5

    240.7
```

This separation of data and program logic radically alters how you as a programmer solve problems.

Overview of APL

What follows is only an introduction to many of the facilities of APL. Detailed descriptions are given in the General Reference Section, beginning on page 27.

APL consists of three fundamentals, **arrays, functions,** and **operators**:

- **Arrays** are ordered rectangular collections of data items.
- **Functions** accept arrays as arguments, and produce new arrays as results.
- **Operators** accept both arrays and functions, applying the functions in special ways, also producing new arrays as results.

IBM's APL Language Reference lists a good example on pages 3 and 4 illustrating many of the fundamentals of the language, including the enhancements of APL2. Appendix N, beginning on page 407 shows their illustration, paraphrased in CAPLIB2 symbols.

APL Arrays

Arrays are the means in APL for organizing and storing data. They have **rank** according to the number of dimensions, or axes. Those consisting of a single item, either a number or character, are called **simple scalars**, and have rank 0. **Vectors** are zero or more items ordered in a row, and have rank 1. A **matrix** has rows and columns, and has rank 2. Higher ranked arrays are just called arrays.

Arrays are named and given data values with the **assignment** function, which is a left arrow (←) in standard APL, and the pound sign (#) in CAPLIB2:

Array items in classic APL could only be either simple scalar numbers or characters, and an array either was all numeric, or all character. In APL2, an array can contain both character and numeric items. An item can even be an array, whose items can be arrays, and so on. Such an array is called *nested*.

```
apl> " a becomes the scalar 3
apl> a # 3
apl> "
apl> " b becomes the vector 1 3 5
apl> b # 1 3 5
apl> "
```

```
apl> " c becomes the character vector 'abc'
apl> c # 'abc'
apl> "
apl> " a becomes a new vector and is displayed
apl> L # a # 2 X a + b

    8 12 16

apl> "
apl> " d becomes a nested vector and is displayed
apl> L # d # ('apples' 'oranges' 'pears'),[.5]a

 apples oranges pears
      8      12    16
```

Here are some rules illustrated by the APL statements above:

- APL statements are evaluated right to left, and there is no function precedence, such as multiplication being done before addition. However there are **binding strengths**, discussed in the next section, and parentheses can be used to make associations explicitly.
- APL arrays can change their data values, and they can change their shape. Above, variable **a** changed from a scalar of rank 0 to a vector or rank 1, dimension 3.
- You can insert "`L#`" almost anywhere in an APL statement to display results to that point.

Binding Strengths

As indicated previously, all functions execute according to their position. **The rightmost function whose arguments are available is evaluated first**. This is the **right-to-left rule**. So, if you're new to APL, and you're used to multiplication taking precedence over addition, you're likely to make the mistake of expecting **2 X 3 + 4** to evaluate to 10 instead of 14.

The **right-to-left rule** doesn't cover all situations. Should **+.X/a b c** be evaluated as **(+.X)/(a b c)** or **(+.X)/a (b c)**, or one of the other possible bindings? In other words, given three names or symbols, how is it determined if the center one is associated with the name or symbol on the left or right? In APL2, a hierarchical set of **binding strength rules** gives the answer. Each of the binding strengths in this table are discussed in a following section:

Hierarchy of Binding Strengths	
Binding Strength	What is Bound
Brackets	Brackets to what is on their left
Specification left	Left arrow to what is on its left
Right operand	Dyadic operator to its right operand
Vector	Array to an array
Left operand	Function to its left argument
Left argument	Function to its left argument
Right argument	Function to its right argument
Specification right	Left arrow to what is on its right.

- Note 1: For binding, the branch right arrow behaves as a monadic function.
- Note 2: Brackets and monadic operatos have no binding strength on the right.
- Note 3: Parentheses override the default binding.

Binding Strengths and Brackets

Brackets have the highest binding strength. They indicate **indexing** (page 185) if the object to their left is an array, and **axis** (page 53) if the object to their left is a function or operator. Given -

```
apl>a#'hat'
apl>b#4 8 10
apl>c#'w' 2 'x' 7
```

then because of binding strength, these two apl statements have the same result:

```
apl>a b c[2]

 hat  4 8 10   2

apl>a b (c[2])

 hat  4 8 10   2
```

Binding Strengths and Specification, Left

After brackets, the binding strength of specification (# , see also page 47) to the object on its left is the next highest:

```
apl>a#2 3 4
apl>a[3]#'H'
apl>a

 2 3 H
```

Given -

```
apl>a#2 3 4
apl>b#8
apl>c#'new'
```

these two statements produce the same result -

```
apl>a b#c

 2 3 4  new

apl>a (b#c)

 2 3 4  new
```

and in both cases **b** would be:

```
apl>b

new
```

Binding Strengths and Right Operand

The **right operand** of a dyadic operator has the third highest binding strength, to its operator. Given -

```
apl>a#2 3RI6
apl>b#3 2RI6
apl>c#2 2RI4
```

Then these two statements product the same answer:

```
apl>+.X/a b c

 106 156
 241 354

apl>(+.X)/a b c
```

```
 106 156
 241 354
```

Note that a right operand is never an array. No primitive dyadic operators take an array right operand. This statement, which attempts to force a dyadic operator to accept an array as right operand produces an error:

```
apl>+.(X/(a b c))

+.(X/(a b c))
  ^ 63 - missing operator/function during execute processing.
```

Binding Strengths and Vector Notation

The fourth highest binding strength is given to **vector binding**, which is vector notation in which two arrays are written next to each other. Here the first two APL statements are equivalent to the second two:

```
apl>2 1 3 / 2 3 4

2 2 3 4 4 4

apl>1 2 + 3 4

4 6

apl>(2 1 3) / (2 3 4)

2 2 3 4 4 4

apl>(1 2) + (3 4)

4 6
```

Binding Strengths and Left Operand

The fifth highest binding strength is given to the **left operand** with its **operator**. Because of its higher binding strength as right operand, **multiply (X)** binds to the **dot (.)** in the expression **+.X/** rather than to the reduce /:

```
apl>+.X/(1 2 3) (4 5 6) (7 8 9)
```

```
 732

apl>+.(X/(1 2 3) (4 5 6) (7 8 9))

+.(X/(1 2 3) (4 5 6) (7 8 9))
  ^ 63 - missing operator/function during execute processing.

apl>(+.X)/(1 2 3) (4 5 6) (7 8 9)

 732
```

Binding Strengths and Left and Right Arguments

Number six in binding strength is the left argument to its function, followed by the binding strenth of the right argument, which is just a restatement of the right-to-left rule:

```
apl>2 3 X 4 5 + 6 7

20 36

apl>2 3 X (4 5 + 6 7)

20 36

apl>(2 3 X 4 5) + 6 7

14 22
```

Binding Strengths and Specification, Right

Lowest in binding strength is **specification** to what's on its right, so that variable **a** in the following expression ends up a vector instead of a scalar:

```
apl>L#a#1 2 3

1 2 3
```

Mixing Data Types

In classic APL, an array could consist of either numeric or character data, not both. An enhancement in APL2 and in CAPLIB2 allows a mixture. Here's a matrix named m with five rows and three columns. Each item in the first row and column, along with the last item in the last row and column, contain character vectors. The rest of the data items are all numeric scalars:

```
      m
food  calories protein
milk       160       9
apple       60       1
bread       75       2
jelly       50    none
```

Simple versus Nested Arrays

If every item in an array is a simple scalar, it's called a *simple* array, which was the only type of structure available in classic APL. Now you can have *nested* arrays, as in matrix `m` shown above. Nested arrays have some new attributes, such as *depth*, *prototype*, and *type*, which are discussed over the next few subsections.

Depth

Depth measures the "deepest" you may go in selecting items of an array before arriving only at a simple scalar. Consider the matrix `m` above. Begin with a *depth counter* set to 0. `m` is not itself a simple scalar, so add 1 to the *depth counter*, and select the vector "food" next for consideration. "food" isn't a scalar either so add 1 again to the *depth counter*, and select "f" from "food". 'f' **is** a simple scalar, so *depth counter*, which now contains 2, shows the *depth* of `m`.

There's a primitive function that produces *depth*, invoked with the *equal underbar* symbol (≡) in APL2, and with the *accented equal* symbol (‘=) in CAPLIB2. For instance the CAPLIB2 expression -

```
apl> ‘= m
```

produces 2 for the matrib `m` shown above.

Display Function

A way of graphically portraying the depth of an array, along with other information, is with a defined function (page 86 begins a description of defined functions) called **display**. It's distributed with IBM's APL2 Program Product. Output is a character matrix showing the data from its array argument contained in a series of nested boxes, with symbols in the borders that indicate the rank of the array, its data types, and possibly whether it's nested.

When you trace your pencil from the outside of the output to an item, without entering any boxes unnecessarily, the number of lines crossed is the depth of that item. The depth of the entire array is equal to the maximum number of lines crossed to an item using this rule. In the followng two examples, the depths are 2 and 3:

```
apl> display 1 2 'more' (3 'a') (2 2RI4) 'b'

+$--------------------------+
|     +$---+ +$----+ +$--+   |
| 1 2 |more| | 3 a | U1 2| b |
|     +----+ ++----+ |3 4| - |
|                    +~--+   |
+E--------------------------+

apl> display 1 2 'more' (3 'or') (2 2RI4) 'b'

+$-----------------------------+
|     +$---+ +$-------+ +$--+   |
| 1 2 |more| |   +$-+ | U1 2| b |
|     +----+ | 3 |or| | |3 4| - |
|            |   +--+ | +~--+   |
|            +E-------+         |
+E-----------------------------+
```

Appendix M contains a list of the source for **display**, written in CAPLIB2 symbols, and a table showing what the symbols in the borders of the output mean.

Types, Prototypes, and Fill Items

Certain operations in APL, such as **disclose**, **expand**, **replicate**, and **take**, produce an expanded structure based on their arguments, and require a **fill item**. Choosing fill in classic APL was easy. It was a blank for character arrays, and 0 for numeric ones. But in APL2, the same array can contain both characters and numerics, so how do you decide what the fill item is?

In APL2, the fill item is the **prototype** of the array, which is defined as the **type** of its first item. The **type** of an array is another array with the same structure and depth, but with a 0 or blank in the same position as each number or character of the original.

The APL2 expression "↑0 ρ ⊂R" gives the **type** of variable R, and "↑0 ρ ⊂ ↑R" gives the **prototype**. The equivalent CAPLIB2 expressions are "Y0R'Zr" and "Y0R'ZYr". The functions used are:

- **first**, ↑ and Y, described on page 135;
- **enclose**, ⊂ and 'Z, described on page 106;
- **reshape**, ρ and R, described on page 291;

Having the fill item be the prototype, which is the type of the first item, means the APL2 interpreter must remember this information anytime an array is reshaped to be empty. After the first of the following two statements, a is empty, but the interpreter must "remember" its prototype before it was made empty in order to satisfy the second statement:

```
apl>a#0RL#a#(2 3)(4 5 6 7)(8 9)

 2 3  4 5 6 7  8 9

apl>Ya

0 0
```

APL Functions

Functions accept arrays as **arguments**, producing new arrays as results. Functions can be **defined, derived, primitive**, or **system**:

- **Defined functions** are defined by you. They're the means by which you "program" in APL.
- **Derived functions** are produced by **operators,** which are described on page 19.
- **Primitive functions,** which are **built in** to APL, are represented in APL statements by symbols, such as **R** for shape and reshape. Primitive functions can be **scalar** or **mixed**.
- **System functions** behave like primitive functions, and you use them syntactically the same. Their names all begin with the quad symbol (□ for APL2, L for CAPLIB2). They all relate to the environment in which you're working. **Lcr** for example gives you the *canonical representation* of a defined function. A table listing all the system functions is on page 317.

Valence

Functions are also categorized according to their **valence**, or number of arguments. **Niladic** functions have no arguments. Only defined functions can be niladic, there being no primitives with this valence. **Monadic** functions accept one argument, and **dyadics** accept two.

For monadics, the function symbol or name is to the left of its argument (R, or *shape*, is described on page 311):

```
apl> R 1 2 3
3
```

For dyadics, the function symbol or name is between its arguments (R, or *reshape*, is described on page 291):

```
apl> 3 4 R 1 2 3
```

R above represents both monadic **shape** and dyadic **reshape**. It's called **ambi-valent**, and the symbols used for all APL primitive functions have this attribute. The context determines which is intended. As the statements are scanned from right to left, in both cases above, first the vector "1 2 3" is recognized, and then **R** is encountered. Whether this is monadic **shape** or dyadic **reshape** is determined by whether a left argument is next derived.

Shared Variables

IBM's APL uses shared variables to communicate with programs not part of the APL environment. These are not implemented in CAPLIB2.

Scalar Functions

Scalar functions apply to their arguments item-by-item, regardless of their structure. If they're dyadic (accepting two arguments), they generally expect the arguments to be of identical rank and shape, except if one argument is a single item, either a scalar or one-element vector, it will be replicated to match the number of items and shape in the other argument. Generally, scalar functions yield results with shapes identical to their arguments.

The prosaic description in the previous paragraph of what makes a scalar function is formally defined in IBM's manual with, "a function is a scalar function if indexing distributes over it." (Indexing is described in the Reference Section, on page 185.) They're referring to the *distributive property* introduced in algebra, where if a, b, and c are real numbers, then $a(b+c) = ab + ac$. The rules are:

- Function F is monadic scalar, given any array r and vector i if $(Fr)[i] \Leftrightarrow Fr[i]$ ($\Leftrightarrow$ signifies "means the same as").
- Function F is dyadic scalar, given conforming arrays l and r, and vector i if $(lFr)[i] \Leftrightarrow Fr[i]$.

There are only two scalar functions that accept character data arguments. They are the comparisons **equal** and **not equal**. A complete list of all APL scalar functions is listed in Appendix D. Here are some examples:

```
apl> "addition      apl> "subtraction   apl> "multiplication
apl> 1 2 3 + 4 5 6  apl> 1 2 3 _ 4 5 6  apl> 1 2 3 X 4 5 6

     5 7 9               -3 -3 -3            4 10 18

apl> "floor         apl> "ceiling       apl> "reciprocal
apl> D -3.14 2.718  apl> S -3.14 2.718  apl> % 4 -5

    -4 2                 -3 3                0.25 -0.2

apl> 'abc' = 'axc'  apl> 'abc' ^= 'axc'

    1 0 1                0 1 0
```

Mixed Functions

There can be two reasons why they might be called mixed functions. First, where they're dyadic, the two arguments don't have to agree either in shape or in data type. Usually the left argument is numeric, giving instructions for manipulation of the right argument, which can either be numeric, character, or any mixture thereof. A complete list of all APL mixed functions is listed in Appendix C. Here are some examples:

```
apl> "reshape                apl> " shape
apl> L # a # 2 3  R  1 2     apl> R a

    1 2 1                       2 3
    2 1 2

apl> "index generator        apl> "index of
apl> I 5                     apl> 'abcdef' I 'cad'

    1 2 3 4 5                   3 1 4
```

```
apl> "take                  apl> "drop
apl> 3 Y 2 4 6 8            apl> 2 U 2 4 6 8

   2 4 6                       6 8
```

Operators

Operators accept both arrays and functions as **operands**, applying the functions in special ways to produce **derived functions**, which produce new arrays as results. IBM's "APL2 Programming: **Language Reference**" indicates you can also use **defined operators**, but I couldn't figure out how they work, so they're not implemented in CAPLIB2.

There are five operators:

- **Reduction:** Denoted by a slash (/), reduction accepts as its left argument a scalar dyadic function, and as its right argument a numeric array. It applies the function between adjacent items in the array. If **v** is a vector `+/v` produces the sum of all elements in **v**.
- **Scan:** Denoted by a backslash or slope (\ ), **scan** is closely related to the reduction operator. It also accepts as its left argument a scalar dyadic function, and as its right argument a numeric array. The result is equivalent to the catenation of the results of applying the scalar dyadic function using **reduction** over successively longer selections from the right argument. The expression `X \ 2 3 4` produces the result 2 6 24, where 2 is the result of `X/2`, 6 is from `X/2 3`, and 24 is from `X/2 3 4`.
- **Axis:** Operators such as **scan** and **reduction** described above are first defined for vectors, and then applied to arrays of higher rank as though they were sets of vectors defined along a particular dimension, or axis. For example, a matrix has two dimensions, rows and columns, and can be considered as a set of vectors either running along each row or down each column. The **axis** operator is denoted by brackets enclosing an expression which yields the index of the dimension along which the vectors are defined. For example, if **m** is a matrix, `+/[1]m` denotes reduction down each column, and `+/[2]m` reduction across each row:

```
apl> L # m # 2 3 R I 6    apl> +/[1]m     apl> +/[2]m
```

```
 1 2 3                          5 7 9           6 15
 4 5 6
```

- **Inner Product:** The dot (.) denotes inner product. Its arguments are two scalar dyadic functions and two arrays. One form, `m+.Xn`, is commonly called the **matrix product** of **linear algebra**:

```
apl> L # m # 2 2 R 2 3 4 5    apl> L # n # 2 3 R 5 + I6

    2 3                            6  7  8
    4 5                            9 10 11

apl> m +.X n

    39 44 49
    69 78 87
```

- **Outer Product:** Accepting a primitive scalar dyadic function and two arrays as arguments, outer product applies the function between all possible combinations of items from the left array argument with items from the right array argument:

```
apl> 1 2 3 J.X 1 2 3 4 5

    1 2 3  4  5
    2 4 6  8 10
    3 6 9 12 15
```

- **Each:** CAPLIB2 and APL2 only. Accepts a function and either one or two arrays as arguments, depending on whether the function is dyadic or monadic: If dyadic, **each** applies the function to corresponding pairs of items of the two arrays; If monadic, the function is applied to each item of the array. For details, see the reference section on page 103.

Workspaces

When you create named arrays and defined functions interactively in APL, they're stored in memory in what is called the **active workspace**. Numerous workspaces can be stored on disk, any one of which you can load into memory as the active workspace. You can modify variables and defined functions while the workspace is in memory, then save it back on disk. There are 16 APL System Commands, all prefixed with a right parenthesis, for manipulating workspaces and their content. See Appendix E for a table of all the System commands.

Data Constants

Constants in APL can be scalars or vectors consisting of character (see page 72) or numeric (see page 21) data. Numeric constants can be integers, fractional real or complex numbers. The following statement assigns to variable **a** a vector consisting of two integers (1 and 400,000), a real fractional number (2.3), a complex number ($6i7$), and a 3-element character vector (**'abc'**):

a # 1 4e5 2.3 6j7 'abc'

Vector Notation

In classic APL, you could produce a vector by justaposing scalar constants, as in -

```
apl> R a # 1 3 9 11 32
5
```

APL2 extends this with **Vector Notation**, where if you juxtapose a mixture of two or more arrays or constants in an expression, the result is a vector:

```
apl> R 2 6 'd' 4 'w'
5
apl> a # 1 2 3
apl> L # d # 'abc' a

 abc  1 2 3

apl> Rd

2

apl> b # 3 4RI12
apl> L # c # a b

 1 2 3   1  2  3  4
```

```
         5  6  7  8
         9 10 11 12

apl> Rc

2
```

You control nesting in Vector Notation by grouping items with parentheses and quote marks:

```
apl> L # a # (1 2 3) (4 5 6) (7 8)

 1 2 3  4 5 6  7 8

apl> R a

3

apl> L # b # 'red' 'white' 'blue'

 red white blue

apl> R b

3

apl> L # c # (9 7 4) 'box' (7 'f' 9 'g')

 9 7 4  box   7 f 9 g

apl> R c

3

apl> L # d # ('up' 'up') 'and' 'away'

  up up   and away

apl> R d

3
```

Number Display

There are several ways you can control the display of numbers in CAPLIB^2 (see Print Precision on page 260, Comparison Tolerance on page 77, and Format on page 139). Unless you tell it otherwise, CAPLIB^2 makes a best effort to display numbers in a reasonable fashion. Most obviously, it eliminates leading zeros to the left of the decimal point and trailing zeros to the right of the decimal point. Slightly more subtle is its printing reasonable numbers of significant digits. With π for example, you have to stop somewhere to the right of the decimal. CAPLIB^2 by default does so after seven digits (see page 257 for a description of the **pi times (O)** function):

```
apl>O1

3.1415927
```

You can override by altering **Print Precision** (see page 260):

```
apl>Lpp#5
apl>O1

3.1416
```

Or, you can tell the interpreter explicity how many significant digits using the **format** function (see page 139), subject to the limitations of the computer architecture:

```
apl>15 13 F O1

3.1415926535898
```

CAPLIB^2 will print numbers in scaled form according to a set of rules that are slightly different from IBM's APL2:

- Any number that needs more than 15 positions to print in conventional form will be printed in scaled notation:

  ```
  apl> 10 X 12345678901234 123456789012345

  123456789012340     1.23457e15
  ```

- Any fractional number for which the number of digits left of the decimal is zero, and the number of zeros to the right of the decimal point before the first significant digit exceeds **Lpp**, will be printed in scaled notation:

```
apl>L#a#1.23456 0.1234 0.12345 0.000123 0.0000123

1.23456 0.1234 0.12345 0.000123 0.0000123

apl>Lpp#4
apl>a

1.235 0.1234 0.1235 0.0001        1.23e-5
```

Alignment of Columns of Numbers

With IBM's APL2, decimal points, the E in scaled notation, and the J for complex numbers align in columns. CAPLIB2 doesn't do quite as good a job. It'll align decimal points in an array of purely real numbers, but when complex numbers are involved, it simply ensures the columns line up left justified:

```
apl>3 2 R 1 2.3 1.2 12.34 0.1 5.6

1.0  2.30
1.2 12.34
 .1  5.60

apl>2 4 R 1 12.3 345 6j7 .1 .12 1j2 16j6

1.0   12.30   345   6j7
.1    0.12    1j2   16j6
```

Character Vector Display

Simple character vectors are printed contiguously, and nested arrays are indented:

```
apl>L#a#'f' 'a' 'c' 'e'

face

apl>‘=a

1

apl>L#c#'f' 'a' 'ce'
```

```
 fa ce

apl>‘=c

2
```

Array Display

A simple matrix is displayed as a rectangular plane:

```
apl>3 4 R I 12

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

Arrays of more than two dimensions are displayed as though they were collections of two-dimensional rectangular planes, with numbers of blank lines interspersed to indicate shape. A three-dimensional array, for example, has a blank line separating each successive rectangular plane:

```
apl>2 3 4 R I 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
```

For arrays of more than three dimensions, an additional blank line separates planes of each dimension, so that one blank line separates planes in the third lowest dimension, two blank lines separate planes in the fourth lowest dimension, etc:

```
apl>2 2 2 3 R I 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
```

Nested Array Display

Nested arrays are displayed indented one space, and nested items within an array are displayed with an additional preceding and succeeding space. In the next example, a nested array is formatted, and APL functions are used to replace the resulting spaces with carets:

```
apl>b#F(1 2 3) (4 5) 6 (7 8 9 10)
apl>b[(' '=b)/IRb]#'^'
apl>b

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

Nested character items in a column that also contains numeric items are right-justified:

```
apl>c#Rb#F4 2R'one' 'two' 1111 22 -4 5 7'?'
apl>b[(' '=b)/IRb#,b]#'^'
apl>cRb

^^one^two^
^1111^^22^
^^^-4^^^5^
^^^^7^^^?^
```

Chapter 2

General Reference Section

2.1 Introduction

Everything in the General Reference Section is arranged in alphabetic order, from APL scalar dyadic function **AND** thru the *System Command* **)WSID**. Each section begins with a heading line with the name being described, in large boldface letters, at the right margin. If the name involves an APL symbol, it will appear to the right of the name on the heading line, along with its CAPLIB2 equivalent. For example, the section on the aforementioned **AND** function lists the APL symbol ∧ and its CAPLIB2 equivalent &.

ACCOUNT INFORMATION ⎕AI **Lai**

Quad ai is an APL2 system variable not implemented in CAPLIB2. In APL2, it yields a vector of four items:

- **Lai**[**1**]: User identification.
- **Lai**[**2**]: Compute time.
- **Lai**[**3**]: Connect time.
- **Lai**[**4**]: User response time..

AND ∧ &

Returns 1 or 0 depending on the logical *and* of left and right, which're considered to be either “true” or “false”. A “true” value is one whose absolute value is greater than or equal to **comparison tolerance Lct**, and a “false” is any value which isn't “true.”

APL Usage

```
0 0 1 1 & 0 1 0 1

    0 0 0 1
```

C Usage

CAPLIB2 ubroutines **and, andp, dbland, dblandp, iand**, and **iandp** are related, differing only in the data types of arguments accepted, and the value returned. They all, however, implement the **logical and** operation: Both arguments must be “true” for the returned value to be “true”.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     And; Andp; Dbland; Dblandp; Dyadic; Endoper;
     Iand; Iandp; Idyadic; Litvect; Quadout; Relative;
     double left=1.0,rite=1.0,ret;
     int ileft=1,irite=1,iret;
     #include "aplinit.h"

     endoper(quadout(relative(and,
          litvect("0.0 0.0 1.0 1.0"),
          litvect("0.0 1.0 0.0 1.0"))));
     endoper(quadout(dyadic(dbland,
          litvect("0.0 0.0 1.0 1.0"),
          litvect("0.0 1.0 0.0 1.0"))));
     endoper(quadout(idyadic(iand,
          litvect("0 0 1 1"),
          litvect("0 1 0 1"))));
     andp(&left,&rite,&iret);
     printf("andp returned %d\n",iret);
     printf("dbland returned %f\n",dbland(left,rite));
     dblandp(&left,&rite,&ret);
     printf("dblandp returned %f\n",ret);
     printf("iand returned %d\n",iand(ileft,irite));
     iandp(&ileft,&irite,&iret);
```

```
     printf("iandp returned %d\n",iret);
}

0 0 0 1
0 0 0 1
0 0 0 1
andp returned 1
dbland returned 1.000000
dblandp returned 1.000000
iand returned 1
iandp returned 1
```

See Also

Comparison Tolerance (**Lct**), nand, nor, and not.

APLCB

Aplcb.h describes the structure representing APL variables. If stdio.h also included, aplcb.h should come after.

```
#if !defined(APLCB_INCL)
#define APLCB_INCL
#define APLFUNC_IN_USE 2048 /* Function definition in use. */
#define APLINDX 1024 /* Variable is used for indexing. */
#define APLGROUP 512 /* Variable describes APL group. */
#define APLFUNC 256 /* Variable describes a function. */
#define APLLABEL 128 /* Variable describes a label. */
#define APLCPLX 64 /* Variable consists of complex numbers. */
#define APLAPL 32 /* Nested variable. */
#define APLCBTM 16 /* reserved */
#define APLMASK 78 /* APLCHAR+APLINT+APLNUMB+APLCPLX */
#define APLCHAR 8 /* Variable consists of characters. */
#define APLINT  4 /* Variable consists of integers. */
#define APLNUMB 2 /* Variables consists of real numbers. */
#define APLTEMP 1 /* Variable is temporary. */
#define ERR 0
#define MAXMSG 100 /* Maximum error message length. */
#define NOTOK 0
#define OK 1
#define APL_REAL (APLINT + APLNUMB)
```

```
#define APL_NUMERIC (APL_REAL + APLCPLX)
typedef union apluptr Apluptr;
union apluptr {
     double *apldata; /* data ptr */
     int    *aplint;
     char   *aplchar;
     struct aplcb **aplapl;
     void   *aplvoid;
};
typedef struct aplcb *Aplcb;
struct aplcb {
     Apluptr aplptr;
     int aplflags;
          /*   1 = Temporary variable.       */
          /*   2 = Double float variable.    */
          /*   4 = Integer variable.         */
          /*   8 = Character variable.       */
          /*  16 = Temporary aplcb.          */
          /*  32 = Aplcb pointer variable.   */
          /*  64 = Not presently used.       */
          /* 128 = APL label in function.    */
          /* 256 = APL function definition.  */
          /* 512 = Group name.               */
          /*1024 = Index variable.           */
          /*2048 = Function def. in use.     */
     int aplcount; /* # els. in var. */
     int aplrank; /* 0=scalar 1=vector 2=matrix, etc. */
     int *apldim; /* Dimension pointer. This points to the */
          /* first byte beyond field APLSIZE.  The memory  */
          /* for the APLCB and the array pointed to by     */
          /* APLDIM are obtained in a single unit.         */
     int aplsize; /* Size of data item in characters. */
};
#endif
```

APLCHAR.H

Aplchar.h is included automatically when you specify APLMAIN on the INCLUDES statement (see page 178) in your C program. It defines a character array which maps the PC keyboard to the APL character set. It contains these items:

- **#define APLCHNO 133:** This is the number of APL symbols mapped to the PC keyboard.
- **int aplchno = APLCHNO:** Globally accessible variable containing a count of APL symbols mapped to the PC keyboard.
- **int aplmaxln = 2:** Globally accessible variable containing the maximum element length in array **aplchar** described below.
- **char *aplchar[APLCHNO] = { " ", "!",...,"∼" }:** Each item in this array maps a particular APL symbol to a sequence of one or more keys on the PC keyboard. For example, the *iota* symbol, ⍳ , maps to the uppercase I. **Aplchdef.h**, described on page 31, defines the indices for all the mappings. See also the table of CAPLIB2 symbols on page 350. A table listing what the pointers in **aplchar** point to is contained in Appendix B.

APLCHDEF.H

Aplchdef.h contains **#define** statements such as

#define DOWNSTILE 36

for every possible index to global array **aplchar**, described on page 30. See also Appendix B.

APLCOMEX

Aplcomex is a CAPLIB2 subroutine that offers a common interface for processing both *compress* and *expand* operations in your C programs:

C Usage

```
#define INCLUDES APLCB+APLCHDEF+APLMAIN+STDIO+APLDERIV
#include "includes.h"
main()
{
     Aplcomex; Compress; Dervfree; Endoper; Expand;
     Indxsub; Litvect; Newderiv; Perm;
     Reshape; Quadout; Temp;
     Aplderiv dp;
     Aplcb tmp;
```

```
     #include "aplinit.h"

     dp = newderiv(NULL,NULL,NULL); /* New aplderiv. */
     dp->deriv_flags |= DERPERM; /* Mark it permanent. */
     dp->deriv_func = aplcomex; /* Does compress and scan. */
     dp->deriv_left.func.wax = compress;
     dp->deriv_left.cb = perm(litvect("1 0 1 0 1"));
     endoper(quadout(tmp=perm(
          dp->deriv_func(dp,NULL,indxsub(5)))));
     dp->deriv_left.func.wax = expand;
     endoper(quadout(dp->deriv_func(dp,NULL,temp(tmp))));
     dp->deriv_flags &= ~DERPERM; /* It's ok to free. */
     dervfree(dp); /* Also frees deriv_left.cb */
}

1 3 5
1 0 3 0 5
```

See Also

Compress (/), Expand (\ ), aplderiv.h.

APLCOPY

C program usage only, copies an apl variable.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Aplcopy; Endoper; Litvect; Quadout;
     Aplcb a,b;
     #include "aplinit.h"

     a = quadout(litvect("1.0 3.14 9.999 10.0"));
     b = quadout(aplcopy(a));
     endoper(a);
     endoper(b);
}

1 3.14  9.999 10
1 3.14  9.999 10
```

See Also

Aplcb.h.

APLDEBUG.H

Apldebug.h is included in your C program when you add APLDEBUG to your #define INCLUDES statement (see page 178). It has a single #define statement in it, setting C Preprocessor variable APL_DEBUG to either 0 or 1. It is used in 10 CAPLIB2 routines, to decide whether to generate memory checking statements: aplfree; aplinit; aplmall; errinit; errstop, funcexec; funcexed; funcexee; and funcsusp.

See Also

Includes.h, Appendix H, which describes memory debugging.

APLDERIV.H

Many of the CAPLIB2 subroutines expect pointers to structures of type **aplderiv**. As indicated in the tutorial section on page 19, APL **operators** accept both arrays and functions as **operands**, applying the functions in special ways to produce **derived functions**, which produce new arrays as results. A structure of type **aplderiv** is used to represent such a derived function, and helps to control processing.

The **aplderiv** structure is defined by adding the APLDERIV keyword on the **#define INCLUDES** statement at the head of your C program, as in this example:

```
#define INCLUDES APLCB+APLMAIN+STDIO+APLDERIV
#include "includes.h"
main()
{
...
```

Here's the C declaration:

```
struct aplderiv {
     int            deriv_flags;
     int            deriv_axis_int;
     double         deriv_axis_dbl;
```

```
    Aplcb           deriv_axis_cb;
    void           *deriv_op;
    Aplcb          (*deriv_func)(Aplderiv, Aplcb, Aplcb);
    Deriv_sub       deriv_left;
    Deriv_sub       deriv_rite;
};
```

- **deriv_flags**: Most CAPLIB^2 subroutines will free the space occupied by an aplderiv structure passed as an argument, unless it is marked as "permanent" by doing a logical OR of deriv_flags and **DERPERM** (defined when you specify **APLDERIV** on the **#define INCLUDES** statement at the head of your C program). At this time there are no other significant flags, so deriv_flags will be either **DERPERM** or 0.

- **deriv_axis_int**: For those routines that require only a single integer for an axis value, this field fills the bill. The same value is repeated in deriv_axis_dbl, and as the first item in the aplcb structure (see page 29) pointed to by deriv_axis_cb. If deriv_axis_int is set to -1, it means the default should be taken for axis.

- **deriv_axis_dbl**: Routines that need a single fractional real number for an axis use this field, the integer part of which is replicated in deriv_axis_int.

- **deriv_axis_cb**: Points to an aplcb structure for one or more axis values, the first of which appears in deriv_axis_dbl.

- **deriv_op**: Set by CAPLIB^2 subroutine **slashtra** to point to the structure which identifies the token representing the current operator to be either *slash*, *slash bar*, *slope*, or *slope bar*. It's referenced in CAPLIB^2 subroutines **aplcomex, execspel, reducef, scanfrnt, slashtrb,** and **slashtrc,** primarily to determine the default for axis if none was supplied in the expression involving the operator. Subroutine **newderiv**, which allocates new aplderiv structures, sets deriv_op to NULL.

- **deriv_func**: Set by CAPLIB^2 subroutines as follows:
  - By **dottran** to one of three subroutines:
    - **execjot**, which handles outer product when the function to be applied is a primitive scalar dyadic (addition, multiplication, division).
    - **execjot2** to handle other dyadic functions for outer product.
    - **execdot** to handle inner product.

– By **eachtran** to the address of subroutine **each**, to do **each** operator processing.
– By **slashtrb** to the address of subroutine **aplcomex**, which does **reduce** and **compress** processing.
– By **slashtrc** to the address of either subroutine **reducef** to do **reduce** processing, or **scanfrnt** to do **scan** processing.

Deriv_func is referenced in subroutine **eachdyad** to handle the **each** operator when the function to be applied is derived, and in subroutine **derived**, reproduced here to illustrate using the aplderiv structure:

```
/* derived APL2 V1.0.0 *****************
* Called from execmons and execdyan to *
* execute a derived function.          *
***************************************/
#define INCLUDES APLCB+APLDERIV
#include "includes.h"
Aplcb derived(dp,left,rite)
Aplderiv dp;
Aplcb left,rite; /* arguments */
{
     Dervfree;
     Aplcb out=NULL;

     out = dp->deriv_func(dp,left,rite);
     dervfree( dp ); /* free structure. */
     return(out);
}
```

- **deriv_left and deriv_rite**: These elements are substructures of type **Deriv_sub** (see below) that describe the functions the operator applies to produce the derived function, e.g. for the CAPLIB^2 expression $a + .Xb$, which applies **inner product** between arrays **a** and **b**, **deriv_left** will describe the scalar dyadic function **plus** (+), and **deriv_rite** will describe **times (X)**.

Here's the C declaration for **Deriv_sub**:

```
typedef struct deriv_sub Deriv_sub;
struct deriv_sub {
     int            funcode;
     void          *fun;
```

```
    void           *sdp;
    Aplcb           cb;
    deriv_type      type;
    sub_dyad        func;
};
```

- **funcode**: Contains one of a set of codes that describe the function or operator that is an operand to the operator whose entry is stored in **deriv_func**. Acceptable codes are defined when you specify the APLDERIV keyword on the **#define INCLUDES** statement at the head of your C program, as shown previously in this section.
  - **DEFINED_FUNCTION**: User-defined function, handled by funcmain.
  - **DERIVED_FUNCTION**: Operator derived function (scan, each, etc).
  - **LEFT_ARROW**: Assignment.
  - **FUNCTION_TOKEN**: One of the standard APL functions, (plus, times, divide, etc.)
  - **APL_NULL**: Outer product.
  - **DIERESIS**: Each.
  - **DOT**: Inner product.
- **fun**: Points to different types of structures depending on the value of funcode:
  - **DERIVED_FUNCTION**: Aplderiv.
  - **DEFINED_FUNCTION**: Aplfunc (see page 41.
  - **FUNCTION_TOKEN**: Funstruc (see page 155.
  - Otherwise fun is NULL.
- **sdp**: Will be NULL, unless the function represented is a standard, builtin scalar dyadic.
- **cb**: Some operators, compress and expand for examples, expect an array operand instead of a function. Field **cb** will contain the address of an **Aplcb** structure in those cases.
- **type**: Identifies the function type according to the types and numbers of arguments. Here's an excerpt from aplderiv.h:

```
typedef enum
  {NO_TYPE, DIX, WAX, DBX, FLR, MOA, MON, MOX, DOX, MEX,
   DNL, DRV}
  deriv_type;
typedef union {
  Aplcb (*dix)(Aplcb, Aplcb); /* dyad mixed w/o axis */
  Aplcb (*wax)(Aplcb, Aplcb, int); /* compress/expand */
  Aplcb (*dbx)(Aplcb, Aplcb, double); /* laminate */
  Aplcb (*flr)(void*, Aplcb, Aplcb); /* relationals. */
  Aplcb (*moa)(void*, Aplcb); /* monadic scalars */
  Aplcb (*mon)(Aplcb); /* monadic mixed w/o axis */
  Aplcb (*mox)(Aplcb, int); /* monadic mixed w/axis */
  Aplcb (*dox)(Aplcb, Aplcb, Aplcb); /* dyadic w/axes */
  Aplcb (*mex)(Aplcb, Aplcb); /* monadic w/axes */
  Aplcb (*dnl)(Aplcb, Aplcb, Aplcb, char *); /* pick */
  Aplcb (*drv)(Aplderiv, Aplcb, Aplcb); /* derived */
  sub_dyad;
```

- **sub_dyad**: See above.

C Usage

Here are two CAPLIB2 statements, the first of which gives the dot product of two matrices, and the second giving a count of the character "e" in each word of the phrase "Still round the corner there may wait –"

```
apl> (2 3RI6) +.X 3 2RI6

22 28
49 64

apl> text#'Still round the corner there may wait'
apl> +/ `1 'e' = (' ' ^= text) `Z text

 0 0 1 1 2 0 0
```

and here is the equivalent C program:

```
#define INCLUDES APLCB+APLCHDEF+APLMAIN+STDIO+APLDERIV
#include "includes.h"
main()
{
     Cdyadic; Chrvect; Cne; Dervfree; Each; Endoper;
```

```
     Execdot; Indxsub; Litvect; Newderiv; Partitn;
     Perm; Quadout; Reducef; Reshape; Temp;
     void subfill(int, Deriv_sub*);
     Aplderiv dpa,dpb;
     Aplcb text,counts;
     #include "aplinit.h"

     dpa = newderiv(NULL,NULL,NULL); /* New. */
     dpa->deriv_flags |= DERPERM; /* Mark it permanent. */
     dpa->deriv_func = execdot; /* Inner product. */
     subfill(PLUS,  &(dpa->deriv_left));
     subfill(TIMES, &(dpa->deriv_rite));
     endoper(quadout(dpa->deriv_func(dpa,
          reshape(litvect("2 3"),indxsub(6)),
          reshape(litvect("3 2"),indxsub(6))
     )));

     dpa->deriv_func = each;
     subfill(EQUAL,&(dpa->deriv_left));
     text=perm(
       chrvect("Still round the corner there may wait"));
     counts=each(dpa,chrvect("e"),
          partitn(cdyadic(cne,chrvect(" "),text),text,NULL));
     dpb = newderiv(NULL,NULL,NULL); /* New. */
     dpb->deriv_flags |= DERPERM; /* Mark it permanent. */
     dpb->deriv_func = reducef;
     subfill(PLUS,&(dpb->deriv_left));
     dpa->deriv_left.funcode = DERIVED_FUNCTION;
     dpa->deriv_left.fun = dpb;
     endoper(quadout(
       each(dpa,NULL,counts) /* counts w/b freed */
     ));
     endoper(temp(text)); /* free aplcb */
     dpa->deriv_flags &= ~DERPERM;
     dervfree(dpa); /* frees both dpa and dpb
}

void subfill(code,sub)
int code;
Deriv_sub *sub;
{
     Funexec; Pickdyad;
```

```
     struct apltoken tok;
     Apltoken tokptr;

     tok.token_code = code;
     tok.token_flags=0; /* So funexec won't copy token. */
     tokptr = funexec(&tok); /* Get info. on code. */
     sub->funcode = tok.token_code;
     sub->fun = tok.token_ptr.token_function;
     sub->sdp = pickdyad(sub->fun);
}

22 28
49 64

 0 0 1 1 2 0 0
```

APLENV.H

aplenv.h is included in your C program when include **includes.h** in your C program. It defines two preprocessor variables:

1. APL_DOS: Set to 1 of the environment is DOS, and 0 otherwise.
2. APL_DOS_H: Set to <dos.h> if the environment is DOS, and "apldummy.h" otherwise

See Also

Includes.h

APLEXT.H

Certain variables are needed globally in the APL environment. When programming in C, **Aplext.h** text must be included at the top of your main program to declare them. If you also include **aplcb.h**, then include **aplext.h** after. Here's an itemized list of the contents of **aplext.h**:

- **int aplerr:** When any of the CAPLIB^2 subroutines recognize an error, they will post an appropriate code in this variable. Appendix A contains a complete list of these error codes. If your program decides to handle the error rather than terminate, it should reset **aplerr** to zero before calling any more CAPLIB^2 subroutines.

- **char *aplfile:** When working with the interactive APL interpreter, if you've issued the **)load** command to load a workspace, **aplfile** will point to its name. Also, if you've issued a **)save** command specifying a workspace name, **aplfile** will point to that name.

- **double fuzz:** This is the **comparison tolerance**, referred to by the system variable name **Lct** in APL.

- **int indxorg:** This is the **index origin**, referred to by the system variable name **Lio** in APL.

- **struct treelist *treehdr:** This is a push-down stack used to control localization of variable names during user-defined function execution.

- **void *aplfphd:** Input file stack. Used by the **)run** command to handle nested **)run** commands.

- **char quadav[4]:** Initialized to "Lav," the system variable name used in APL to reference the **atomic vector**.

- **char quadct[4]:** Initialized to "Lct," the system variable name used in APL to reference the **comparison tolerance**.

- **char quadio[4]:** Initialized to "Lio," the system variable name used in APL to reference the **index origin**.

- **char quadlx[4]:** Initialized to "Llx," the system variable name used in APL to reference the **latent expression**.

- **int aplcurs:** Current location in output line.

- **#include "aplchar.h":** See page 30.

- Following are all related to delayed workspace commands. You don't need to know what these are used internally in CAPLIB^2 subroutines for, but you do need to know the names exist.

  - **struct aplcb *aplwkcb:** Workspace command text.
  - **char *aplwkcp:** Pointer into workspace command text.
  - **char *aplwken:** Pointer to end workspace command text.
  - **int aplwkcd:** Workspace command code, OFF, LOAD, etc.

APLFUNC.H

The **aplfunc** structure is defined by adding the APLFUNCI keyword on the **#define INCLUDES** statement at the head of your C program.

```
#if !defined(APLFUNC_INCL)
#define APLFUNC_INCL
#define NILAD 256 /* Function is niladic. */
#define MONAD 512 /* Function is monadic. */
#define DYAD 1024 /* Function is dyadic. */
#define RETVAL 2048
#define MOP 4096
#define DOP 8192
typedef struct aplfunc *Aplfunc;
struct aplfunc {
  struct aplfunc *funcsiqp;
     /* Used for stacking operations.                 */
     /* Execqfxa assigns NULL. No other reference.    */
     /* Funcsiqp m/b defined so funcflag will be in   */
     /* the same relative position as aplflags in     */
     /* an aplcb structure, because avlnodes can point */
     /* to both aplcbs and aplfuncs.                  */
  int funcflag;
     /* Always 256 to indicate fixed function         */
     /* Ref: execexee, execexei, execqfxa, funcmain,  */
     /*      lovfsfn savfsfn.                         */
  int functype;
     /* Ref: apledit, execexei, execnila, execqfxa,   */
     /*      execqfxd execqfxg, funcinit, funcopy,    */
     /*      funcmain, funcopy, lovfsfn.              */
     /*  256 = niladic, F                             */
     /*  512 = monadic, F r                           */
     /* 1024 = dyadic,  l F r                         */
     /* 2048 = result,  z # F                         */
     /*                 z # F r                       */
     /*                 z # l F r                     */
     /*                 z # (lo mop) r                */
     /*                 z # l (lo mop) r              */
     /*                 z # (lo dop ro) r             */
     /*                 z # l (lo dop ro) r           */
     /* 4096 = monadic operator                       */
     /*                 (lo mop) r                    */
```

```
   /*                l (lo mop) r                   */
   /* 8192 = dyadic operator                        */
   /*                (lo dop ro) r                  */
   /*                l (lo dop ro) r                */
int functotl;
   /* Count of tokens in functary.                  */
   /* Ref: apledcl, apledit, aplediv, execexee,     */
   /*      execqfxam expunge, execqfxk, funcopy,    */
   /*      lovfsfn, lovfsfo, lovfsfp, savfsfn,      */
   /*      savfsfo.                                 */
int funcstmt;
   /* Count of statements in function.              */
   /* Ref: apledit, aplediw, execqfxa, execqfxb,    */
   /*      execqfxf, execqfxj, execqfxk, funcopy,   */
   /*      funcexec, funcexef, funcgoto, funcopy,   */
   /*      functrgo, lovfsfn.                       */
int *functokc;
   /* Array of token counters, one per statement.   */
   /* For statements with labels, this count will   */
   /* be reduced by 2.                              */
   /* Ref: apledcl, aplediu, aplediv, aplediw,      */
   /*      apledrc, execqfxa, execqfxd, execqfxf,   */
   /*      execqfxj, execqfxk, expunge, funcexee,   */
   /*      funcopy, lovfsfn, savfsfn.               */
struct aplcb *functext;
   /* Pointer to function text.                     */
   /* Ref: aplcpyc, aplediw, execqcr, execqfxa,     */
   /*      execqfxb, expunge, funcmsg, funcopy,     */
   /*      lovfsfn, savfsfn.                        */
struct apltoken *funcname;
   /* Pointer to function name token.               */
   /* Ref: aplediv, execqfxa, execqfxl, funcexec,   */
   /*      funcmsg, funcopy, functrac, functrgo,    */
   /*      lovfsfp, savfsfn, treeroot.              */
struct apltoken **functokp;
   /* Ptr to array of token headers,                */
   /* one for each statement in the function.       */
   /* The queue of tokens gets altered during       */
   /* statement execution.                          */
   /* Ref: apledcl, apleddl, apledit, aplediu,      */
   /*      aplediv, aplediw, aplediy, apledmd,      */
   /*      aplednw, apledqu, apledrc, execqfxa,     */
```

```
    /*      execqfxb, execqfxd, execqfxf, execqfxj,   */
    /*      execqfxk, expunge, funcexee, funcopy,     */
    /*      lovfsfn, lovfsfp, savfsfn.                */
  struct apltoken *functary;
    /* Memory for all tokens is obtained             */
    /* once. This points to that block of memory.    */
    /* Ref: apledcl, aplediv, execqfxa, execqfxb,    */
    /*      execqfxk, expunge, funcopy, lovfsfn,     */
    /*      lovfsfo, lovfsfp, savfsfn, savfsfo.      */
  struct apltoken *funcvars;
    /* List of local variables.                      */
    /* Ref: execqfxa, execqfxd, execqfxg, execqfxi,  */
    /*      funcinit, funcmain, funcopy, lovfsfp,    */
    /*      savfsfn.                                 */
    /* The variable to contain the returned value, if */
    /* there is a returned value, will be first in   */
    /* this list.  If the function is dyadic, the left*/
    /* operand will be next, followed by the right   */
    /* operand, if the function is either dyadic or  */
    /* monadic.  The rest of the list is the local   */
    /* variables as indicated on the function header. */
  struct apltoken *funclabs;
    /* List of labels.                               */
    /* Ref: aplediw, execqfxa, execqfxf, execqfxi,   */
    /* funcinit, funcopy, lovfsfp, savfsfn.          */
    /* Token_offset will contain the number of the   */
    /* statement with the given label.               */
};
#endif
```

APLINIT

Aplinit.h must be **includ**ed in any main C program you write that's going to use a CAPLIB^2 routine. The **#include** should appear immediately after your last declaration, and before your first executable statement. **Aplinit.h** establishes the APL environment needed by the CAPLIB^2 routines.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
```

```
Dyadic; Plus; Quadout; Scalar;
```

#include "aplinit.h"

```
     endoper(quadout(dyadic(
       plus,scalar(1.2),scalar(3.4))));
}
```

```
4.6
```

APLMEM.H

The file **aplmem.h** is included by adding the APLMEM keyword on the **#define INCLUDES** statement at the head of your C program. It consists of two **#define** statements, changing all references to the **free** and **malloc** standard library functions to **aplfree** and **aplmall**. Of the 748 CAPLIB^2 subroutines, 70 call either **free, malloc**, or both, but they all include **aplmem.h**. This makes it easy to debug problems related to memory overwrites, or attempts to free memory already freed.

```
#define free aplfree
#define malloc aplmall
```

APLMSG

When a C-language program calls a CAPLIB^2 subroutine, errors are signaled back by a nonzero value in external int variable **aplerr**. The C-language program can call **aplmsg** to obtain a character string containing the error message associated with a particular error number. Appendix A lists all the possible error messages.

APL Usage

Messages are printed automatically whenever errors occur.

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Aplmsg; Divide; Dyadic; Endoper; Litvect; Quadout;
```

```
     Aplcb a;
     extern int aplerr;
     #include "aplinit.h"

     a = dyadic(divide,litvect("1.0"),litvect("0.0"));
     if (aplerr) {
          fprintf(stderr,"%s\n",aplmsg(aplerr));
          exit(aplerr);
     }
     endoper(quadout(a));
}

39 - real division by 0.
```

APLTOKCD.H

Contains definitions for special token codes that may appear in field **token_code** of struct apltoken. The QUAD_xx definitions are all negative and contiguous, with QUAD_AI first and QUAD_ZZ last. Token codes 256 and above are reserved for user-defined function types. See aplfunc.h and apltoken.h.

```
#define DEFINED_FUNCTION 256
#define NO_MORE_TOKENS -2
#define VECTOR_TOKEN -3
#define OPERAND_TOKEN -4
#define MESSAGE_TOKEN -5
#define DERIVED_FUNCTION -7
#define AXIS_INDEX_TOKEN -8
#define NULL_TOKEN -9
#define FUNCTION_TOKEN -10
#define QUAD_AI -11
#define QUAD_AV -12
#define QUAD_CR -13
#define QUAD_CT -14
#define QUAD_DL -15
#define QUAD_EX -16
#define QUAD_FX -17
#define QUAD_HT -18
#define QUAD_IO -19
#define QUAD_LC -20
#define QUAD_LX -21
```

```
#define QUAD_NC -22
#define QUAD_NL -23
#define QUAD_PP -24
#define QUAD_PW -25
#define QUAD_RL -26
#define QUAD_TC -27
#define QUAD_TS -28
#define QUAD_TT -29
#define QUAD_UL -30
#define QUAD_WA -31
#define QUAD_EA -32
#define QUAD_FC -33
#define QUAD_TZ -34
#define QUAD_ZZ -99
#define TOKPERM 1
```

APLTOKEN.H

Describes the structure used to tokenize APL statements. Used in subroutine aplparse. The QUAD_xx definitions must remain negative and contiguous, with QUAD_AI first and QUAD_WA last. Token codes 256 and above are reserved for user-defined function types. See aplfunc.h.

```
#if !defined(APLTOKEN_INCL)
#define APLTOKEN_INCL
#include "apltokcd.h"
typedef struct apltoken *Apltoken;
struct apltoken {
  union {
    struct apltoken *token_next_ptr;
      /* Pointer to next token. */
    int token_next_offset;
      /* Offset to next token in workspace. */
  } token_queue;
  int token_code;
      /* VECTOR_TOKEN, OPERAND_TOKEN, LEFT_ARROW,
      DERIVED_FUNCTION */
  int token_offset;
      /* Offset in statement where token was found,
      or for a function definition, the statement number
      if this token is a label. */
```

```
  union {
    char *token_string;
      /* For OPERAND_TOKEN, points to string containing
      the name.  For MESSAGE_TOKEN, points to message.
      This sort of token only occurs during parse. */
    struct aplcb *token_vector;
      /* During initial parse, points to a literal vector. */
    struct apltoken *token_stack;
      /* During execution, axis or index token stack. EXECBRKT
      creates a token with token_code AXIS_INDEX_TOKEN, and
      pushes onto its token_stack field all the indices it
      encounters between the left and right brackets. */
    void *token_function;
      /* In APL2, expressions can evaluate to a function. */
    struct aplderiv *token_deriv;
      /* For token_code == DERIVED_FUNCTION, points to structure
      describing the derived function, e.g. +/[1]. */
  } token_ptr;
  int token_flags;
      /*  1 = permanent token. */
  void *token_work;
      /* May contain e.g. a pointer to a function definition
      during execution.
      EXECEXEE Initializes token_work to NIL for the function/
      operator token most recently pulled from the input.
      EXECEXEI is called by execexee when current token is
      type OPERAND_TOKEN to check if it is a defined function.
      If so, execexei sets token_work to the address of the
      defined function definition structure.
      GETCODE is called to determine operator/function code
      for a token.  If token_work is not NULL, getcode assumes
      the token is for a defined function, and returns code
      DEFINED_FUNCTION (256).
      NEWTOK allocates new tokens.  It initializes token_work
      to NIL. */
};
#endif
```

ASSIGN ←

The left arrow (#), or **specification**, is the means whereby you assign values to APL variables.

APL Usage

Here's a typical APL statement that assigns 6 to the variable *sum*:

```
sum # 1 + 2 + 3
```

Assignment might be used several times in one statement, and by inserting the sequence `L #` in one or more parts of a statement, you can print the result to that point. If assignment wasn't the last (or leftmost), thing done in an APL statement, the final result is automatically displayed:

```
apl> x # 2 + L # 3 X y # 4        apl> x

12                                14

apl> y

4
```

The expression to the right of a specification arrow must produce an array. A variety of objects can be to the left of the arrow, and APL2 allows more than classic APL, which permitted only a name, followed by an optional bracketed set of indices (see **indexing** on page 185). APL2 also allows a parenthesized list of names, or an expression that selects positions from an existing array. Here's an example of a parenthesized list:

```
apl> L#(j k) # 3 4                apl> j

3 4                               3

apl> k                            apl> L#(j k) # 31

4                                 31

apl> j                            apl> k

31                                31
```

Selective Specification

Selective specification replaces selected items of an array. Any expression selecting items from an array may be placed on the left of an assignment arrow. Here's a simple example:

```
apl> v # 10 20 30 40
apl> (2Yv)#100 200
apl> v

100 200 30 40
```

The parenthesized expression operates as though the values in the array being indexed were numbers representing the relative positions of each item. The rightmost name in the expression, ignoring brackets for indexing, is the name of the array to which selective assignment is made.

The rules for how the assignment is completed in selective specification are complicated, and aren't ones I would've selected were I designing the language. To me the KISS (Keep It Simple Stupid!) principle should always take precedence. I'm not sure even IBM understands their own rules. When I tried these assignments in their demo version of APL2, the second one caused it to fail:

A←(2 3)(2 3 4 5)(10 20)
(4 ↑ 2 ⊃ A)←2 3 4

The same two assignments using the APL2 interpreter in CAPLIB^2 produce these results:

```
apl> a#(2 3)(2 3 4 5)(10 20)
apl> (4Y2‘Xa)#2 3 4
apl> b

 2 3   2 3 4  2 3 4  2 3 4  2 3 4   10 20
```

Because IBM's demo version failed in some circumstances such as the one just described, I could not ensure CAPLIB^2 has implemented selective assignment according to the rules intended.

Selective Specification Rules

1. The rightmost name in the expression, ignoring brackets used for indexing, is the name whose value will be altered.

2. **Pick (‘X)** with an empty left argument, and **first** both return a "whole array," contrasted with a "subset of an array," according to IBM, and this alters what items are replaced:

```
apl> a#'abcd'
apl> ((I0)‘Xa)#10 20 30
apl> a

10 20 30

apl> a#b#(2 3)(4 5 6 7)(8 9)
apl> Ya

2 3

apl> (Ya)‘=1‘Xb

1

apl> (1‘Xb)#10 11 12
apl> b

 10 11 12  4 5 6 7  8 9
```

```
apl> 1‘Xb

2 3

apl> (Ya)#10 11 12
apl> a

 10 11 12  4 5 6 7  8 9
```

3. Any selection involving functions other than just first or pick as described above selects a subset of an array:

```
apl> L#a#b#(2 3)(2 3 4 5)(10 20)

 2 3  2 3 4 5  10 20
```

```
apl> (3Y2‘Xa)#'abc'
apl> a

 2 3   abc 5   10 20
```

4. If the item to the right of the left arrow is a scalar, it is paired with each item from what's selected by the expression of the left of the assignment arrow:

```
apl> (Eb)#0
apl> b

 0 0  0 0 0 0  0 0
```

Here are a few more examples of selective specification:

```
apl> L#m#x#3 4R'abcdefghijkl'
```

```
apl> m[2;3]#'L'
apl> m
```

```
abcd
efgh
ijkl

apl> m#x
apl> (2 1Ym)#'OM'
apl> m

Obcd
Mfgh
ijkl

apl> m#x
apl> m[1;2 4]#'GJ'
apl> m

aGcJ
efgh
ijkl

apl> m#x
apl> m[1 3;1 4]#'*'
apl> m

*bc*
efgh
*jk*
```

```
abcd
efLh
ijkl

apl> m#x
apl> (4Y,Tm)#'J*%L'
apl> m

JLcd
*fgh
%jkl

apl> m#x
apl> (,m)#I12
apl> m

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

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
Assign; Chrvect; Endoper; Execute; Litvect;
     #include "aplinit.h"

     assign("left",  litvect("1.0 2.0 3.0"));
     assign("right", litvect("4.0 5.0 6.0"));
     endoper(execute(chrvect("left + right")));
}
```

```
5 7 9
```

See Also

Plus (+).

ATOMIC FUNCTION ☐AF **Laf**

Quad af is an APL2 system variable not implemented in CAPLIB^2. The CAPLIB^2 expression **Lav I r** produces the same answer as the APL2 expression ☐**AF R**.

```
apl> Lio#0
apl> Lav I 'OZ' " Same answer as Laf 'OZ' in APL2

79 90
```

See Also

Atomic Vector (Lav), Index Of (I).

ATOMIC VECTOR ☐AV **Lav**

Quad av is an APL system variable that is a character vector of dimension 256 containing all possible characters. The index of any character, such as "A", is given by **LavI'A'** (see also **Index Of**).

The **atomic vector** is useful, for example, in instances where you want to rank character values. The only direct comparisons you can make of character data in APL are **equal** and **not equal**. The expression (`'abc' < 'def'`) for example isn't permitted, and will generate an error message if tried.

The idea is to maintain machine independence. Because characters may be ranked differently from one machine to the next, a level of indirection is enforced. One technique for ranking character data is to first encode each character string as the decimal value of a $base_{256}$ number derived from each character's position in the **atomic vector**. An example is given below:

APL Usage

```
" here's a sample array of character data
L # k # 3 3 R 'ghiabcdef'
```

```
    ghi
    abc
    def

" each character's position in Lav is given by -
Lav I k

    104 105 106
     98  99 100
    101 102 103

" ordering the rows in ascending sequence -
k[A(R Lav) B Lav I T k;]

    abc
    def
    ghi
```

See Also

Assign (#), Decode (B), Grade (A), Indexing([]), Quad Output (L), Shape (R), System Variables, Transpose (T).

ATTRIBUTES □AT **Lat**

Quad at is an APL2 system variable not implemented in CAPLIB2. In APL2, it yields a vector of attributes about various apl2 objects.

AXIS

Axis and **dimension** mean the same thing. The **shape vector** of an array consists of all its dimensions, or axes. A matrix of three rows by four columns would have a shape vector of **3 4**. The **shape** function (ρ in classic APL, **R** in CAPLIB2 - see page 311) produces the shape vector for any array.

The number of items in any array is the product of its axes, given by the CAPLIB2 expression **+/Ra**, where **a** is the name of the array (see page 275 for an explanation of **reduce**).

An **empty array** has one or more of its dimensions equal to 0.

Axis Specification

Certain functions and operators, indicated by the table below, all can work along vectors, with higher ranking arguments considered to by sets of vectors. The **axis of application** is the dimension along which these sets of vectors lie, and can be explicitly indicated by writing an expression in brackets to the right of the function or operator (Note: An asterisk (*) next to a name in the following table indicates the function did not allow axis specification in classic APL.):

Functions Allowing Axis Specification		
CAPLIB[2] Symbol	**APL2 Symbol**	**Name of Function**
+	+	*Add
&	∧	*And
!	!	*Binomial
O	○	*Circular
‘X	⊃	Disclose
%	÷	*Divide
U	↓	Drop
‘Z	⊂	Enclose
=	=	*Equal
>	>	*Greater
Z	≥	*Not less
<	<	*Less
K	≤	*Not greater
@	⍟	*Logarithm
S	⌈	*Maximum
D	⌊	*Minimum
X	×	*Multiply
P	⍲	*Nand
‘V	⍱	*Nor
∧ =	≠	*Not equal
V	∨	*Or
*	⋆	*Power
∣	∣	*Residue
,	,	Ravel, Catenate, Laminate
‘O T	⌽ ⍉	Reverse, Rotate
‘Q	⌷	*Squad
_	−	*Subtract
Y	↑	Take

Operators Allowing Axis Specification		
CAPLIB² Symbol	**APL2 Symbol**	**Name of Operator**
\ ‘\	\ ⍀	Scan, Expand
/ ‘/	/ ⌿	Reduce, Compress

Consider **reduction**, which reduces each vector it processes to a single value. By default the axis of reduction is the last (called the *column axis,* the next to last being called the *row axis*). An explicit axis is given as a bracketed number following the slash:

```
apl> L#m#3 4RI12

1  2  3  4
5  6  7  8
9 10 11 12

+/[1]m

15 18 21 24
```

```
+/m

10 26 42

+/[2]m

10 26 42
```

To derive a vector from the set determined by the axis value, hold the index for all dimensions other than the axis dimension to a single scalar value, and supply a vector of all possible index values for the axis dimension:

```
apl> L#n#2 3 4RI24

 1  2  3  4
 5  6  7  8
 9 10 11 12

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

apl> +/[1]n

14 16 18 20
22 24 26 28
30 32 34 36

apl> +/[2]n

15 18 21 24
51 54 57 60
```

```
apl> n[I2;1;1]

 1 13

apl> n[1;I3;1]

1 5 9

apl> n[1;1;I4]

1 2 3 4

apl> +/[3]n

10 26 42
58 74 90
```

Axis Specification with Scalar Dyadic Functions

As indicated above, APL2 permits the application of scalar functions along a specified axis. The general form is -

```
z # l f[x] r
```

where the axis indication **x** is a simple scalar or vector of axes, with no redundancies, such that -

```
(R,x) matches (RRl)DRRr
&/xEI(RRl)SRRr produces 1
```

For example:

```
apl> .3 .2 .1 +[1] m # 3 4 R I 12

1.3  2.3  3.3  4.3
5.2  6.2  7.2  8.2
9.1 10.1 11.1 12.1
```

In classic APL, which didn't permit axis specification with scalars, the result would be an error message:

```
apl> .3 .2 .1 +    m # 3 4 R I 12
              ^ 17 - arguments are not conformable.
```

In APL2 the arguments are conformable if -

```
(Rl) matches (Rr)[x]  or (Rr) matches (Rl)[x]
when x‘=x[Ax] (i.e. x is in ascending order)
```

The shape of the result is the shape of the array with greater rank, and the order axes appear in doesn't affect the result:

```
apl> k#2 3R.1XI6
apl> j#2 3 4RI24
apl> j+[1 2]k

 1.1  2.1  3.1  4.1
 5.2  6.2  7.2  8.2
 9.3 10.3 11.3 12.3

13.4 14.4 15.4 16.4
17.5 18.5 19.5 20.5
```

```
21.6 22.6 23.6 24.6

apl> (j+[1 2]k)‘=j+[1 2]k

1
```

See Also

Assign (#), Catenate (,), Compress (/), Expand (\ ), Index Generator (I), Indexing ([]), Laminate (,), Quad Output (L),Reverse (‘O), Rotate (‘O).

BINOMIAL ! !

For positive integer arguments, m ! n is the number of distinct ways m things can be selected from n things. For positive non-integer values m ! n is equivalent to the CAPLIB^2 expression (! n) % (! m) X (! n _ m) .

Determining how to implement the binomial function for CAPLIB^2 was difficult because IBM's Manual ("APL2 Programming: Language Reference Version 2 Release 1," SRL SH21-1061-00) is incomplete in its description, and has several mistakes. For L and R both positive integers and $R > L$, it gives the expression -

$$(\,!\,R) \div (\,!\,L) \times -L \tag{2.1}$$

as equivalent to L ! R. The correct expression is:

$$(\,!\,R) \div (\,!\,L) \times\ !\,R - L \tag{2.2}$$

For L and R both negative integers and $R < L$, IBM's manual gives the expression -

$$(-1 \star R - L) \times (+1)\ !\ (\,|\ \ L+1) \tag{2.3}$$

The correct expression is:

$$(^{-}1 \star R - L) \times (-R+1)\ !\ -L+1 \tag{2.4}$$

I used these rules for binomial:

- If either argument is not an integer (i.e. fractional or complex number), or if both arguments are positive integers and $left < right$ use formula 2.2 above.

- If the arguments are equal or left is 0, return 1.
- If both arguments are negative integers and $left < right$ use formula 2.4 above.
- Otherwise return 0.

The above rules produce answers consistent with those in IBM's manual, except for the single case involving complex numbers (see below).

APL Usage

```
apl> 2!5

10

apl> 2 3 4 ! 6 18 24

15 816 10626

apl> 3 ! .05 2.5 -3.6

 .01543769 0.3125 -15.456

apl> a # -6 + I11
apl> a J.! a

   1  -4   6 -4  1 0 0 0 0 0  0
   0   1  -3  3 -1 0 0 0 0 0  0
   0   0   1 -2  1 0 0 0 0 0  0
   0   0   0  1 -1 0 0 0 0 0  0
   0   0   0  0  1 0 0 0 0 0  0
   1   1   1  1  1 1 1 1 1 1  1
  -5  -4  -3 -2 -1 0 1 2 3 4  5
  15  10   6  3  1 0 0 1 3 6 10
 -35 -20 -10 -4 -1 0 0 0 1 4 10
  70  35  15  5  1 0 0 0 0 1  5
-126 -56 -21 -6 -1 0 0 0 0 0  1

apl> 2 ! 3j2 " IBM's APL2 produces the answer 1J5.

-1j3
```

```
apl> 0 1 2 3 ! 3

    1 3 3 1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Aplmsg; Binom; Binomp; Binomx; Dyadic;
     Endoper; Ibinom; Ibinomp; Idyadic; Litvect; Quadout;
     double left[2],rite[2],ret[2];
     int ileft,irite,iret;
     #include "aplinit.h"

     endoper(quadout(dyadic(binom,
          litvect("2 2.1"),
          litvect("5 5.6"))));
     *left = 2.1;
     *rite = 5.6;
     binomp(left,rite,ret);
     printf("binomp returned %f\n",*ret);
     *left = 2.0;
     *(left + 1) = 0.0;
     *rite = 3.0;
     *(rite + 1) = 2.0;
     binomx(left,rite,ret);
     printf("binomx returned %fj%f\n",*ret,*(ret+1));
     endoper(quadout(idyadic(ibinom,
          litvect("2 3 4"),
          litvect("6 18 24"))));
     ileft = 2;
     irite = 5;
     ibinomp(&ileft,&irite,&iret);
     printf("ibinomp returned %d\n",iret);
}
```

```
10 13.484871
binomp returned 13.484871
binomx returned -1.000000j3.000000
15 816 10626
```

```
ibinomp returned 10
```

BOOLEAN FUNCTIONS

The Boolean functions in APL are **and, nand, or, nor,** and **not**, so called because they deal only in values representing **TRUE** and **FALSE**. Standard APL represents **TRUE** with the integer 1, and **FALSE** with 0. CAPLIB[2] uses 0 for **FALSE** but considers any non-zero value to be **TRUE**.

```
apl> ~ 0 1 2 3.4 -5 " not (~)

1 0 0 0 0

apl> a J.& a#0 1 2 " and (&)

0 0 0
0 1 1
0 1 1

apl> a J.P a#0 1 2 " nand (P)

1 1 1
1 0 0
1 0 0

apl> a J.V a#0 1 2 " or (V)

0 1 1
1 1 1
1 1 1

apl> a J.‘V a#0 1 2 " nor (‘V)

1 0 0
0 0 0
0 0 0
```

See Also

And, nand, or, nor, and **not.**

BRANCH → $

Defined functions can have numerous APL statements in them, normally executed sequentially. You can change this order by inserting **branch** statements, each comprised of a **right arrow** (this is a dollar sign ($) in the CAPLIB2 symbol set) followed by an expression yielding a scalar or vector.

An empty vector means no branch will occur. Execution continues with the next sequential statement. Otherwise, the integer part of the leftmost or only value designates the next statement to execute, according to these rules:

- All statements in a defined function are sequentially numbered, the **function header** being 0, then next statement 1, and so on.
- If the integer derived from the branch statement is negative, 0, or greater than the largest statement number in the defined function, execution of the function terminates.

Conditional and Unconditional Branches

Using the compress function (/), you can set up branches which are taken according to the truth or falseness of one or more expressions. Here are some common and useful forms:

Common Branch Forms	
$ $(condition)/0$	If the condition is true, then end execution of the defined function. Else fall down to the next sequential statement.
$ $(cond_1 ... cond_n)/label_1 ... label_n$	If any **cond**ition is true, then branch to the label that corresponds to the first true condition. Else fall down to the next sequential statement.
$ $label \, \mathrm{X} \, condition$	If the condition is true, then branch to the specified label. Else end execution of the defined function.

See Also

Defined Function, Labels.

CANONICAL REPRESENTATION ⎕CR Lcr

Quad cr is a monadic system function used to obtain a representation in character matrix form, called *canonical* for reasons I've never understood, of a defined function.

APL Usage

```
apl> Lfx 2 7R'z#sum x z#+/x '

sum

apl> Lcr 'sum'

z#sum x
 z#+/x

apl> sum 1 3 5 7 9

    25
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Execute; Execqcr; Execqfx;
     Litvect; Quadout; Reshape;
     #include "aplinit.h"

     endoper(
       quadout(
         execqfx(
           reshape(
             litvect("2 7"),
             chrvect("z#sum x z#+/x ")))));
     endoper(
       quadout(
         execqcr(chrvect("sum"))));
     endoper(execute(chrvect("sum 1 3 5 7 9")));
}
```

```
sum

z#sum x
 z#+/x

25
```

See Also

Assign (#), Defined Function, Function Establishment (Lfx), Plus (+), Reduction (/).

CATENATE , ,

The **catenate** symbol is the comma. $z\,\texttt{\#}\,l,r$ combines two variables into one by juxtaposing their values, assigning the result to **z**. In this case $RRz \Longleftrightarrow$, S / $(RRl),(RRr),1$.

An optional axis operator determines the axis along which values are juxtaposed. See page 53 for a detailed explanation of axes. The default axis is the last.

Two arrays are *conformable* for catenation along axis I if they differ in rank by at most 1, with their shapes agreeing after dropping dimension I from both if their ranks equal, or from the array with higher rank if not. The exception to this rule is that a scalar or one-element vector is made to be conformable by replication.

In classic APL, Non-empty numeric and character arguments can't be catenated with one another. In APL2 and CAPLIB2, this is no problem, resulting in a nested array.

APL Usage

```
apl> m # 3 3 R 'abcdefghi'
apl> c # '*'
apl> m,[1] v

abc
def
ghi
pqr
```

```
apl> v # 'pqr'
apl> m,[2] v

abcp
defq
ghir

apl> c,(c,[1] m,[1] c),c
```

```
apl> m,v

abcp
defq
ghir

apl> L#z#2 4 6,1 3 5

2 4 6 1 3 5

apl> Rz

6

apl> L#k#2 3RI6

1 2 3
4 5 6

apl> L#h#k,q

1 2 3 7  8
4 5 6 9 10

apl> L#m#2,3

2 3

apl> L#n#2 3

2 3

apl> m ‘= n

1

apl> L#f#’to’ ’ken’

 to ken

apl> Rf
```

```
*****
*abc*
*def*
*ghi*
*****

apl> L#z#’abc’,1 2 3 4

 abc 1 2 3 4

apl> Rz

7

apl> L#q#2 2R7 8 9 10

7  8
9 10

apl> L#x#9 8 7,6 5 4

9 8 7 6 5 4

apl> L#q#9 8 7 6 5 4

9 8 7 6 5 4

apl> x ‘= q

1

apl> L#e#’to’, ’ken’

token

apl> Re

5

apl> ‘=e
```

```
2

apl> `=f

2

apl> L#a#3 4R'blueshoefoot'

blue
shoe
foot

apl> L#b#3 5R'berrylacesstool'

berry
laces
stool

apl> L#z#a,b

blueberry
shoelaces
footstool

apl> Rz

3 9

apl> L#d#2 1R('aquinas' 'more')('ockham' 'shakespeare')

        aquinas more
  ockham shakespeare

apl> `=d

3

apl> c,d

 thomas          aquinas more
 william   ockham shakespeare
```

```
1

apl> L#c#2 1R'thomas' 'william'

 thomas
 william

apl> `=c

2

apl> 's',2 4R'prigtray'

sprig
stray

apl> (2 2 3RI12),'*'

  1  2  3 *
  4  5  6 *

  7  8  9 *
 10 11 12 *

apl> L#u#'sat'

sat

apl> L#v#'teammazerail'

teammazerail

apl> u,v

satteammazerail

apl> L#w#'1: ' '2: '

 1:  2:

apl> y#'log on' 'log off'
```

```
apl> ‘=c,d

3

apl> Yj#'',I0

0

apl> ' ' = Yk#(I0),''

1

apl> L#a#3 4R'bathbeatbind'

bath
beat
bind

apl> a,[1]'x'

bath
beat
bind
xxxx

apl> L#h#'words'

words

apl> L#c#2 2 3RI12

 1  2  3
 4  5  6

 7  8  9
10 11 12

apl> L#d#2 3 3R_I18

 -1  -2  -3
 -4  -5  -6
```

```
apl> L#y#,[I0]y

 log on
 log off

apl> L#g#w,y

 1:  log on
 2:  log off

apl> Rg

2 2

apl> ‘=g

2

apl> 0,[1]2 5RI10

0 0 0 0  0
1 2 3 4  5
6 7 8 9 10

apl> L#b#2 4R'zoomzero'

zoom
zero

apl> a,[1]b

bath
beat
bind
zoom
zero

apl> L#k#2 5R'strawberry'

straw
berry
```

```
 -7  -8  -9

-10 -11 -12
-13 -14 -15
-16 -17 -18

apl> c,[2]d

  1   2   3
  4   5   6
 -1  -2  -3
 -4  -5  -6
 -7  -8  -9

  7   8   9
 10  11  12
-10 -11 -12
-13 -14 -15
-16 -17 -18

apl> L#z#q,[1]s

  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15

 -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 -32 -33 -34 -35
-36 -37 -38 -39 -40
-41 -42 -43 -44 -45

apl> Rz

4 3 5
```

```
apl> h,[1]k

words
straw
berry

apl> L#q#3 5RI15

 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15

apl> L#s#3 3 5R_I45

 -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 -32 -33 -34 -35
-36 -37 -38 -39 -40
-41 -42 -43 -44 -45

apl> L#z#q,[2]s

  1   2   3   4   5
 -1  -2  -3  -4  -5
 -6  -7  -8  -9 -10
-11 -12 -13 -14 -15

  6   7   8   9  10
-16 -17 -18 -19 -20
-21 -22 -23 -24 -25
-26 -27 -28 -29 -30

 11  12  13  14  15
-31 -32 -33 -34 -35
-36 -37 -38 -39 -40
-41 -42 -43 -44 -45
```

```
apl> Rz

3 4 5
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Cat; Chrvect; Endoper; Litvect; Perm; Quadout;
     Reshape; Temp;
     Aplcb c,m;
     #include "aplinit.h"

     c=perm(quadout(chrvect("#"))); /*----------->   #*/
     m=perm( /*---------------------------------> ABC*/
          quadout( /*                             DEF*/
               reshape( /*                        GHI*/
                    litvect("3 3"),
                    chrvect("ABCDEFGHI")
               ) /*                               ABC*/
          ) /*                                    DEF*/
     ); /*                                        GHI*/
     endoper(quadout(cat(m,m,1))); /*---------->   ABC*/
     /*                                           DEF*/
     /*                                           GHI*/

     endoper(quadout(cat(m,m,2))); /*---------->ABCABC*/
     /*                                        DEFDEF*/
     /*                                        GHIGHI*/

     endoper( /*------------------------------> #####*/
          quadout( /*                           #ABC#*/
               cat( /*                          #DEF#*/
                    c, /*                       #GHI#*/
                    cat( /*                     #####*/
                         cat(
                              c,
                              cat(
                                   m,
```

```
                                   c,
                                   1
                              ),
                              1
                         ),
                         c,
                         2
                    ),
                    2
               )
          )
     );
     endoper(temp(c));
     endoper(temp(m));
}
```

See Also

Assign (#), Axis ([]), Reshape (R).

CDYADIC = ∧ =

There are only two scalar dyadic functions for character data: *equal* (=) and *not equal* (∧ =).

APL Usage

```
apl> 'abc' = 'axc'                apl> 'abc' ^= 'axc'

1 0 1                             0 1 0
```

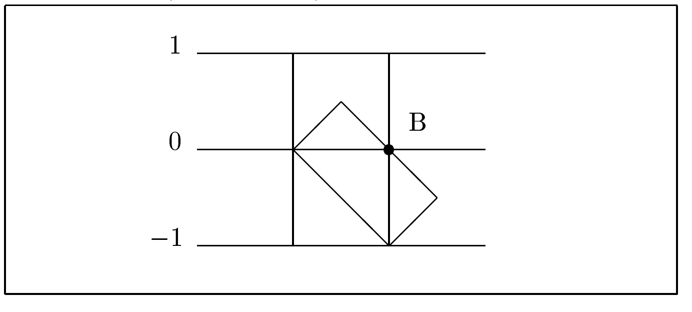
C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
Aplmsg; Cdyadic; Ceq; Chrvect; Cne; Endoper; Quadout;
     #include "aplinit.h"

     endoper(quadout(cdyadic(ceq, /* -----------> 1 0 1 */
          chrvect("abc"),
```

```
          chrvect("axc"))));
     endoper(quadout(cdyadic(cne, /* -----------> 0 1 0 */
          chrvect("abc"),
          chrvect("axc"))));
}
```

CEILING ⌈ S

Upstile (S) when used monadically yields the smallest integer not less than its argument for real numbers. For complex numbers it's more complex : − >. The figure below represents the complex number plane, the horizontal axis being of course the real part, and the vertical the imaginary part. Point **B** is the **ceiling** for any complex number falling in the slanted rectangle, or on one of its right edges. The rectangle has dimensions *square root of* 2 *by square root of* .5 (1.414 × .707):



APL Usage

```
apl> S -3.14 2.718                apl> S -2.7 3 .5

    -3 3                          -2 3 1

apl> S1j2 1.2j2.5 -1.2j-2.5       apl> S1.5j2.5

1j2 1j3 -1j-2                     1j3
```

C Usage

In most environments there is no **ceil** routine in CAP2.LIB. Instead, CAPLIB2 relies on the ANSI Standard **ceil** routine being available. The CAPLIB2 source however does include a ceil.c for those cases where it's missing.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Ceil; Ceilp; Ceilx; Endoper; Litvect; Monadic; Quadout;
     double num[2],ret[2];
     #include "aplinit.h"

     endoper(quadout(monadic(ceil,
          litvect("3.14 -3.14"))));
     *num = -3.14;
     ceilp(num,ret);
     printf("ceilp returned %f\n",*ret);
     *num = 1.5;
     *(num+1) = 2.5;
     ceilx(num,ret);
     printf("ceilx returned %fj%f\n",*ret,*(ret+1));
}

4 -3
ceilp returned -3.000000
ceilx returned 1.000000j3.000000
```

See Also

Floor.

CEQ = =

Compares character data. Returns 1 if equal, 0 if not equal.

APL Usage

```
apl> 'abc' = 'axc'

1 0 1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
```

```
{
     Aplmsg; Cdyadic; Ceq; Chrvect; Endoper; Quadout;
     #include "aplinit.h"

     endoper(quadout(cdyadic(ceq, /* ------------> 1 0 1 */
          chrvect("abc"),
          chrvect("axc"))));
}
```

See Also

CDYADIC, CNE.

CHARCODE

C program usage only. Call **charcode** with a string representing an APL symbol and it will return the integer code used in the CAPLIB2 programming environment for that character. If the argument string isn't a CAPLIB2 representation of an APL symbol, **charcode** returns -1.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Charcode;
     #include "aplinit.h"

     printf("The code for 'R' is %d\n",charcode("R"));
}
```

```
The code for 'R' is 50
```

See Also

CODECHAR

CHARACTER CONSTANTS

Character constants in APL are simply strings of characters surrounded in single quotes.

APL Usage

```
apl> " assign to x a scalar character constant.
apl> x # 'a'

apl> " assign to x a vector character constant.
apl> " to get a single quote, key two quotes.
apl> L # x # 'Can''t you see?'

Can't you see?
```

See Also

Assign (#), Constants, Quad Output (L).

CHRVECT

C program usage only. Call **chrvect** with a character string argument and it will return the address of an APLCB for an APL character variable. See the programming example for the **cdyadic** subroutine on page 69.

See Also

Assign (#), CDYADIC, ENDOPER.

CIRCULAR O **O**

In classic APL the circle symbol (O), used dyadically, invoked a family of fifteen related functions. In APL2 it's expanded to twenty five:

Where functions aren't monotonic, such as the inverse trigonometric (arc) functions, only the principal value is returned.

APL Usage

```
apl> 1  O  (O  1) % 2 3 4

1 0.8660254038 0.7071067812
```

$(\_\,a)\,\mathrm{O}\,b$	a	$a\,\mathrm{O}\,b$
$(1\,\_\,b * 2) * .5$	0	$(1\,\_\,b * 2) * .5$
$\arcsin(b)$	1	$\sin(b)$
$\arccos(b)$	2	$\cos(b)$
$\arctan(b)$	3	$\tan(b)$
$(-1 + b * 2) * .5$	4	$(1 + b * 2) * .5$
$\mathrm{asinh}(b)$	5	$\sinh(b)$
$\mathrm{acosh}(b)$	6	$\cosh(b)$
$\mathrm{atanh}(b)$	7	$\tanh(b)$
$\_\,a\,\mathrm{O}\,b$	8	$\_\,(-1\,\_\,b * 2) * .5$ for $b \geq 0$
		$(-1\,\_\,b * 2) * .5$ for $b < 0$
b	9	Real part of b
$+b$	10	$\mid b$
$0j1\,\mathrm{X}\,b$	11	Imaginary part of b
$*\,0j1\,\mathrm{X}\,b$	12	Phase R

Table 2.1: Dyadic Circular (O) Functions

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Circular; Dyadic; Endoper; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(dyadic(circular,
          litvect("-2"), /* inverse cosine */
          litvect("1 0.7071067812 0"))));
}
```

```
0 0.7853982 1.5707963
```

See Also

Divide (%), Pi Times (O), and Appendix O, on page 411, which describes the trigonometric functions in detail.

CODECHAR

C program usage only. Call **codechar** with an integer ranging from 0 to 106 and it will return the address of the character string representing the corresponding APL symbol in the CAPLIB2 environment.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Codechar;
     #include "aplinit.h"

     printf("The APL symbol for code 50 is '%s'\n",
       codechar(50));
}

The APL symbol for code 50 is 'R'
```

See Also

CHARCODE

)CLEAR

Resets everything in memory to the way it was when you first entered APL from the system prompt.

APL Usage

```
)clear
```

See Also

Appendix E

CNE $\neq$ $\wedge =$

Compares character data. Returns 1 if not equal, 0 if equal.

APL Usage

```
apl> 'abc' ^= 'axc'

0 1 0
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Aplmsg; Cdyadic; Chrvect; Cne; Endoper; Quadout;
     #include "aplinit.h"

     endoper(quadout(cdyadic(cne, /* -------> 0 1 0 */
          chrvect("abc"),
          chrvect("axc"))));
}
```

See Also

CDYADIC, CEQ.

COMMENT ⍝ "

Any statement beginning with a lamp symbol, ("), also known as **cap null**, is considered a comment. APL2 additionally allows you to append a comment to any other statement:

APL Usage

```
apl> " this is a comment. the next statement adds 1 and 2
apl> 1 + 2

3

apl> 1 + 2 " and this too is a comment.

3
```

COMPARISON TOLERANCE ⎕CT **Lct**

The **comparison tolerance** is a system variable used to control comparisons of fractional real numbers. For example, subroutine **gt**, which accepts two arguments of type **double**, returns 1 if the left argument is greater than the right argument plus **comparison tolerance**, and returns 0 otherwise.

Comparison tolerance is necessary because most computers don't precisely represent many fractional real numbers. Usually, .1 is represented by 0.99999..., the number of nines to the right of the decimal point varying from one computer to the next.

APL Usage

In APL the comparison tolerance is referred to by the system variable name **Lct**. You can assign a new value to **Lct** if you wish. It is initialized to $1e-13$. The following illustrates why **comparison tolerance** is necessary:

```
apl> 10=10X+/10R%10

1

apl> Lct#0
apl> 10=10X+/10R%10

0
```

C Usage

In a C program using CAPLIB2 library routines, comparison tolerance is referred to by the name **fuzz**, which must be declared **double** and **extern**.

See Also

Reduce (/), Reshape (R), Times (X).

COMPRESS ⌿ \ ‘/ /

If the left argument is boolean (consisting solely of 1s and 0s), / represents dyadic **compress**, which eliminates elements from its right argument according to the pattern of 1s and 0s (0s indicating elements to eliminate) in its left argument. If the left argument isn't boolean / represents dyadic **replicate**, documented on page 287.

An optional axis operator determines the axis along which values are eliminated (see page 53 for a detailed explanation of axes). The default axis is the last for / and the first for ‘/.

Arguments are **conformable** for **compress** if the length of **lo** in **lo / r** and the dimension of r along which compression occurs agree, except that a scalar or one-element vector for a left argument, or a scalar for a right argument will be extended as needed.

The rank of the result equals the rank of the right argument, except a scalar right argument gives a vector result.

Compress is known as a mixed function because its right argument may be either character or numeric.

APL Usage

```
apl> 1 0 1 1 0 / 2 3 5 7 11

2 5 7

apl> ( c ^= ' ' ) / c # 'this is an example'

'thisisanexample'

apl> L # q # 3 4 R 'abcdefghijkl'

abcd
efgh
ijkl
```

```
apl> 1 0 1 / [ 1 ] q

abcd
ijkl

apl> 1 0 1 / 5

5 5
```

```
apl> 1 0 1 ‘/ q

abcd
ijkl

apl> 0 1 1 0 / q

bc
fg
jk
```

Selective Specification

Compress can be used for selective specification, also described on page 48:

```
apl> L#m#3 2RI6

1 2
3 4
5 6

apl> L#m#3 2RI6

1 2
3 4
5 6

apl> (1 0 1/[1]m)#t
apl> m

 a b
 3 4
 c d
```

```
apl> (1 0 / m)#'abc'
apl> m

 a 2
 b 4
 c 6

apl> L#t#2 2R'abcd'

ab
cd
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Compress; Endoper; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(compress(
          litvect("1 0 1 1 0"),
          litvect("2 3 5 7 11"),
          1))); /* 3rd argument is axis */
}

2 5 7
```

See Also

Assign (#), Axis ([]), Expand (\ ), Quad Output (L), Replicate (/).

CONJUGATE + +

Monadic *conjugate* yields its argument with the real parts unchanged, but imaginary parts negated.

APL Usage

```
apl> + 3 -4 5

3 -4 5

apl> +4 2.3 -3 -.7

4 2.3 -3 -0.7

apl> 1j2X+1j2

5
```

```
apl> +-4

-4

apl> +1j2

1j-2
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Conjug; Conjugp; Conjugx; Endoper; Iconjug;
     Iconjugp; Imonadic; Endoper; Litvect; Monadic;
     Quadout;
     double rite[2],ret[2];
     int irite,iret;
     #include "aplinit.h"

     endoper(quadout(monadic(conjug,
          litvect("3.14 -3.14"))));
     endoper(quadout(imonadic(iconjug,
          litvect("-1 0 1 2 3"))));
     *rite = -5.6;
     conjugp(rite,ret);
     printf("conjugp returned %f\n",*ret);
     *rite = 1.0;
     *(rite+1) = 2.0;
     conjugx(rite,ret);
```

```
     printf("conjugx returned %fj%f\n",*ret,*(ret+1));
irite=-1;
     iconjugp(&irite,&iret);
     printf("iconjugp returned %d\n",iret);
}

3.14 -3.14
-1 0 1 2 3
conjugp returned -5.600000
conjugx returned 1.000000j-2.000000
iconjugp returned -1
```

See Also

Negative (_), Plus (+).

CONSTANTS

Constants in APL can be scalars or vectors consisting of character data, numerics, or a mixture of both. Character constants are described on page 72, numeric constants on page 21, and mixes of characters and numerics on pages 14 and 13.

See Also

Assign (#), Quad Output (L).

CONVECT

C program usage only, creates an apl variable from an array of constants. CONVECT is declared as -

struct aplcb *convect(void *pdata, int cnt, int typ);

Pdata is a pointer to an array of constants of type char, int, or double.
Cnt is a count of items pointed to by *pdata*.
Typ is one of three integer values according to the data type:

- 8 = char
- 4 = int
- 2 = double

See Also

The description of **decode** contains a sample program using *CONVECT*. See also ENDOPER.

)COPY

)copy wsname obj1 obj2 ···

The indicated objects (functions, variables, etc.) are copied from the workspace named "*wsname*." into the active workspace. If the list is omitted *all* objects are copied (other than system variables). An object with the same name as one already in the active workspace replaces it. Workspaces on disk are named **wsname.apl**, but you should leave off the **.apl** on the)**copy** command. You can however qualify the name with its directory path, e.g.)**copy \cap\handbook\wsname a b c**.

See Also

Appendix E, pcopy.

DEAL ? ?

The **deal** function, (`m ? n`), yields a vector containing `m` pseudo-random selections without replacement from population (`I n`) (see the **Index Generator** function). Both arguments must be either scalars or one-element vectors.

APL Usage

```
apl> " randomly select 3 items out of 5
apl> 3  ?  5

2 1 3

apl> " obtain a random permutation of 5 things
apl> 5  ?  5

1 5 2 3 4
```

```
apl> " the index origin determines what is generated
apl> Lio  #  0
apl> 5 ? 5

1 0 3 2 4
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
Deal; Endoper; Iscalar; Quadout;
     #include "aplinit.h"

     endoper(quadout(deal(iscalar(3),iscalar(9))));
}


2 3 5
```

See Also

Assign (#), Index Origin (Lio).

DECODE ⊥ B

For vector arguments, **decode** yields the value of the right argument evaluated in a number system with radices defined in the left argument. If items of **rarg** are nonnegative integers less than scalar **larg**, then **larg B rarg** gives the base-10 equivalent of a base-**larg** number:

```
apl> > 2 B 1 1 1 1

15
```

Decode is extended to arrays by applying each of the radix vectors along the last axis of the left argument to each of the vectors along the first axis of the right argument. If either of these axes is of length 1, it will be extended by replication as necessary to match the length of the other argument:

```
apl> (L#l,l,l # 2 1R2 10) B L # r # 3 2R1 4 0 3 1 2

1 4
0 3
1 2

 2  2  2
10 10 10

  5  24
101 432

apl> (L#    l # 2 1R2 10) B L # r # 3 2R1 4 0 3 1 2

1 4
0 3
1 2

 2
10

  5  24
101 432
```

The shape of the result of the APL statement `(a B b)` is equal to `((-1 U R a),1 U R b)`, and the rank `(0 S _ 1+ R R a)+(0 S _ 1+ R R b)`.

Polynomial Evaluation and Decode

For scalar **larg** and vector **rarg**, **larg B rarg** evaluates a polynomial arranged in descending order of powers with coefficients **rarg** at **larg**, e.g. here's examples of evaluating $x^2 + 2x + 1$ at $x = 3$, and $x^3 + 2x^2 + 3x + 4$ at $x = 1 + i$:

```
apl> 3 B 1 2 1

16

apl> 1j1 B 1 2 3 4

5j9
```

APL Usage

```
apl> " 1 hour 2 minutes 3 seconds is how many total seconds?
apl> 24 60 60  B  1 2 3

3723
```

apl>" evaluate the polynomial $2x^2 - 3.5x + 4$ at $x = 5$

```
apl> 5  B  2 -3.5 4

36.5
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Convect; Decode; Endoper; Iscalar; Litvect; Quadout;
     Reshape; Transpos;
     static int radices[9] = {24,60,60,10,10,10,2,2,2};
     static int values[9] = {1,2,3,1,7,6,1,0,1};
     static int dims[2] = {3,3};
     static int tran[2] = {2,1};
     #include "aplinit.h"

     endoper(
          quadout(
               decode(
                    iscalar(10),
                    litvect("1 7 7 6")
               )
          )
     ); /* --------------------------------> 1776 */
     endoper(
          quadout(
               decode(
                    reshape(
                         convect(dims,2,APLINT),
                         convect(radices,9,APLINT)
                    ),
                    transpos(
```

```
                          convect(tran,2,APLINT),
                          reshape(
                               convect(dims,2,APLINT),
                               convect(values,9,APLINT)
                          )
                     )
                )
           )
     ); /* ---------------------------> 3723 4026 3601
                                         123  176  101
                                          11   24    5 */
}
```

See Also

Drop (U), Encode (N).

DEFINED FUNCTION

A **defined function** is created by you, and is the means whereby you "program" in APL. It consists of a **function header** followed by one or more APL statements. Here's one called **plus** that returns the sum of its two arguments:

```
apl> " see FUNCTION ESTABLISHMENT for Lfx
apl> Lfx 2 15 R 'res # l plus r res # left + right    '

plus

apl> " see CANONICAL REPRESENTATION for Lcr
apl> Lcr 'plus'

res # l plus r
res # l + r

apl> 1 + 2 + 3                    apl> 1 + 2 plus 3

6                                 6
```

Function Time Stamp

IBM's APL2 associates a time stamp that identifies when a defined function was last fixed in the workspace. This is not implemented in CAPLIB2.

Ambi-valent Defined Functions

IBM's APL2 permits defined functions to be **Ambi-valent**, i.e. to be invoked either monadically or dyadically. CAPLIB2 doesn't allow this. A defined function is invoked either niladically, monadically, or dyadically, according to the form of its function header.

See Also

Appendix J, Branching, Canonical Representation, copy, pcopy, Function Editing, Function Establishment, Function Header, Function Stop, Function Trace, Labels, Names(page 233), State Indicator.

DEFINED FUNCTION EXECUTION

You can use a defined function in context where you use a standard APL primitive function, such as **plus (+), minus(_)**, or **times (X)**, but it can't be applied with any of the primitive operators **scan, reduction, inner product,** or **outer product**. This type of operation is allowed in IBM's APL2, but isn't implemented in CAPLIB2:

```
⎕FX'Z← A PLUS B' 'Z←A+B'
PLUS / ⍳3
6
```

A defined function can include statements that invoke other defined functions, and can also recursively call itself.

See Also

Defined Function, Function Stop, Function Trace,)SI.

DEFINED OPERATOR

IBM's manual on APL2 describes "defined operators," which are not implemented in CAPLIB2.

DELAY ☐DL Ldl

Quad dl is a monadic system function in standard APL used to delay execution for a specified period. It has not been implemented in CAPLIB2.

DEPTH ≡ ‘=

Monadic **Depth** reports the level of nesting for its argument, the "deepest" you may go in selecting items before arriving only at a simple scalar.

```
apl>‘=5               apl>‘=’a’          apl>‘=2 2RI4

0                     0                  1

apl>‘=3 2 4 5RI120    apl>Rb#’jim’ ’al’ ’ev’

1                     3

apl>‘=b               apl>‘=‘1b          apl>Rc#’ab’ 1 2 3

2                      1  1  1           4

apl>‘=c               apl>‘=‘1c

2                      1  0  0  0

apl>Rd#’one’ ’two’ (’buckle’ (’my’ ’shoe’))

3

apl>display d

+$-----------------------------------------+
| +$--+ +$--+ +$-------------------------+ |
| |one| |two| | +$-----+ +$------------+ | |
| +---+ +---+ | |buckle| | +$-+ +$---+ | | |
|             | +------+ | |my| |shoe| | | |
|             |          | +--+ +----+ | | |
|             |          +E------------+ | |
|             +E-------------------------+ |
+E-----------------------------------------+
```

```
apl>‘=d             apl>‘=‘1d

4                    1  1  3

apl>‘=‘1‘1d

  0  0  0    0  0  0    1  2

apl>‘=I0            apl>YI0             apl>‘=’’

1                   0                   1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Depth; Endoper; Iscalar; Litvect; Quadout;
     #include "aplinit.h"

     endoper(
          quadout(
               depth(
                    iscalar(10)
               )
          )
     ); /* --------------------------------> 0 */
}
```

See Also

The tutorial section on page 14.

DISCLOSE ⊃ ‘X

APL2 and CAPLIB2 operator only, did not exist in classic APL, **depth** removes one level of nesting. For the expression **zzz # ‘Xrrr**, these statements are true:

- All items of **rrr** must be scalars or arrays of the same rank, though not necessarily the same shape.
- The shape of the result **rrr** is equivalent to -

```
(Rrrr),YS/(R‘1(,rrr),‘ZYrrr)~‘ZI0
```

- The rank of the result **rrr** is equivalent to -

```
(RRrrr)+YS/R‘1R‘1(,rrr),‘ZYrrr
```

- If items of **rrr** have different shapes, each is padded to a shape that's consistent with the greatest length of each dimension for all items of **rrr**. Each item's fill item is used for padding.
- Disclose is the left inverse of enclose, i.e. **rrr** is always given by this expression:

```
‘X‘Zrrr
```

An optional, bracketed vector of axes may be specified. For the expression **zzz # ‘X** [**xxx**] **rrr** these statements are true:

- The number of items in **xxx** must be equal to the highest rank of items in rrr, given by the expression -

```
S/ER‘1R‘1rrr
```

- The values of **xxx** must be nonnegative integers no larger than the sum of the highest rank in **rrr**, plus the rank of **rrr** itself, i.e. contained in the vector given by the expression -

```
I(RRrrr)+S/ER‘1R‘1rrr
```

- The following expression always produces 1:

```
+/((Rzzz)[,xxx])=YS/(R‘1(,rrr),‘ZYrrr)~‘ZI0
```

- The rank of **zzz** is equal to this expression:

```
(RRrrr)+S/ER‘1R‘1(,rrr),‘ZYrrr
```

- The order of axes values in **xxx** affects the shape of the result.
- If all items of **rrr** are scalars, **xxx** must be empty.
- Disclose with axis is the left inverse of enclose with axis, i.e. **rrr** is always produced by this expression:

```
‘X[xxx]‘Z[xxx]rrr
```

- The following two expressions are equivalent:

```
‘Xrrr
‘X[(RRrrr)+IRRYrrr]rrr
```

APL Usage

```
apl> ‘X(2 3 4) (5 6)

2 3 4
5 6 0

apl> L#r#2 3R(I4)’abcd’ ’****’(5 6 7 8)’efgh’ ’HHHH’

 1 2 3 4  abcd ****
 5 6 7 8  efgh HHHH

apl> Rr        apl> L#z#‘Xr        apl> Rz        apl> ‘=z

2 3             1 2 3 4           2 3 4          1
                a b c d
apl> ‘=r        * * * *

2               5 6 7 8
                e f g h
                H H H H

apl> L#e#(2 4RI8) 9 (3 2R’abcdef’)        apl> L#n#‘Xe

 1 2 3 4   9  ab                           1 2 3 4
 5 6 7 8      cd                           5 6 7 8
              ef                           0 0 0 0

                                           9 0 0 0
```

```
                                            0 0 0 0
                                            0 0 0 0

                                            a b
                                            c d
                                            e f

apl> Rn       apl> ‘=n      apl> d#’wheel’ ’of’ ’fortune’
                            apl> L#v#‘Xd
3 3 4         1
                             wheel
                             of
apl> Rv       apl> ‘=v       fortune

3 7           1

apl> L#h#’abcd’ (1 2 3 4) ’wxyz’        apl> L#z#‘X[1]h

 abcd  1 2 3 4  wxyz                     a 1 w
                                         b 2 x
                                         c 3 y
apl> Rz        apl> ‘=z                  d 4 z

4 3            1

apl> L#w#‘X[2]h       apl> Rw        apl> ‘=w

 a b c d              3 4            1
 1 2 3 4
 w x y z

apl> L#z#2 4R’pa’ ’ma’ ’we’ ’by’ ’it’ ’up’ ’on’ ’hi’

 pa ma we by
 it up on hi

apl> L#r#(2 4RI8) (2 4R’abcdefgh’) (z)

 1 2 3 4   abcd    pa ma we by
 5 6 7 8   efgh    it up on hi

apl> L#b#‘X[1 2]r       apl> Rb        apl> ‘=b
```

```
 1 a pa                  2 4 3          2
 2 b ma
 3 c we
 4 d by

 5 e it
 6 f up
 7 g on
 8 h hi

apl> L#y#‘X[1 3]r    apl> Ry    apl> L#v#‘X[2 3]r    apl> Rv

  1  2  3  4         2 3 4       1  2  3  4         3 2 4
  a  b  c  d                     5  6  7  8
 pa ma we by
                                 a  b  c  d
  5  6  7  8                     e  f  g  h
  e  f  g  h
 it up on hi                    pa ma we by
                                it up on hi

apl> )read \my\cap2\help\display.txt
apl> Lfx file

display




apl> display r

+$-----------------------------------------+
| +$------+ +$---+ +$--------------------+ |
| U1 2 3 4| Uabcd| U +$-+ +$-+ +$-+ +$-+ | |
| |5 6 7 8| |efgh| | |pa| |ma| |we| |by| | |
| +~------+ +----+ | +--+ +--+ +--+ +--+ | |
|                  | +$-+ +$-+ +$-+ +$-+ | |
|                  | |it| |up| |on| |hi| | |
|                  | +--+ +--+ +--+ +--+ | |
|                  +E--------------------+ |
+E-----------------------------------------+
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Compute; Disclose; Endoper; Quadout;
     #include "aplinit.h"

     endoper(
          quadout(
               disclose(
                    compute(chrvect("(2 3 4) (5 6)")),
                    NULL
               )
          )
     ); /* -------------------------------> 2 3 4 */
        /*                                  5 6 0 */
}
```

See Also

Depth (‘=), Enclose (‘Z).

DIVIDE ÷ %

Divide implements the standard arithmetic function, except that `(0 % 0)` is defined to produce 1.

APL Usage

```
apl> 1 % 2 4            apl> 3%2        apl> 9 4 7 10 % .25

0.5 0.25                1.5             36 16 28 40

apl> 0j12%4             apl> 0%5        apl>0%0

0j3                     0               1

apl> .3 5 1%0j1 -2 1    apl> 5%0

0j-0.3 -2.5   1         5%0
```

```
^ 39 - real division by 0.
```

C Usage

Subroutine **divide** accepts arguments of type **double**, and returns a value of type **double**.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Divide; Dividep; Dividex; Dyadic; Endoper;
     Idivide; Idyadic; Iscalar; Quadout; Scalar;
     double left[2],rite[2],ret[2];
     #include "aplinit.h"

     endoper(quadout(dyadic(
       divide,scalar(1.0),scalar(4.0))));
     endoper(quadout(idyadic(
       idivide,iscalar(50),iscalar(10))));
     *left = 2.1;
     *rite = 5.6;
     dividep(left,rite,ret);
     printf("dividep returned %f\n",*ret);
     *left = 0;
     *(left + 1) = 12.0;
     *rite = 4.0;
     *(rite + 1) = 0;
     dividex(left,rite,ret);
     printf("dividex returned %fj%f\n",*ret,*(ret+1));
}
```

```
 .25
5
dividep returned 0.375000
dividex returned 0.000000j3.000000
```

See Also

IDIVIDE.

)DROP

)drop wsname

The named workspace **wsname** is deleted from disk. Workspaces on disk are named **wsname.apl**, but you should leave off the **.apl** on the **)drop** command. You can however qualify the name with its directory path, e.g. **)drop \cap\handbook\wsname** under DOS.

See Also

Appendix E

DROP ↓ U

The **Drop** function **(s U v)** can best be understood by first considering the simpler case where **s** is a non-negative scalar integer and **v** is a vector. The result is a vector consisting of the rightmost **(0 S (R v) _ s)** elements of **v**:

```
apl> 2 U 1 2 3 4 5
```

```
3 4 5
```

For the CAPLIB2 expression -

```
z # q U v
```

these statements are all true:

- The rank of the result equals the length of the left argument, or is equal to this expression:

  ```
  (R,q)SRRv
  ```

- If the left argument is negative, elements are dropped from the end instead of the front:

  ```
  apl> -2 U 1 2 3 4 5
  ```

  ```
  1 2 3
  ```

- If **v** is an array, **q** must be a vector with one element for each dimension of **v**. The **i**th element of **q** determines what is to be **drop**ped along the **i**th dimension of **v**:

```
apl> 2 3 U 3 4 R 'abcdefghijkl'

l
```

- The number of subarrays dropped doesn't affect the rank of the result, which is given by this expression:

```
0S(Rv)_|q
```

- The **i**th element of **q** can be zero, in which case no subarrays are removed from the **i**th axis.
- If the **i**th element of **q** exceeds the **i**th dimension of **v**, the result has an **i**th axis of zero length.
- For scalar **v**, **q** can have any length.
- **Drop** doesn't affect the depth of any item of **q**. The depth of **z** is less than or equal to the depth of **q**, except when **q** is a simple scalar.

APL Usage

```
apl> 3 U 12 31 45 10 57

10 57
```

```
apl> -3 U 12 31 45 10 57

12 31
```

```
apl> L#a#3 5R'striperodeplant'

strip
erode
plant
```

```
apl> 1 2 U a

ode
ant
```

```
apl> b # 'storefirstmighthater'
apl> b # b , 'sheettheremetroerase'
apl> L # b # 3 4 5 R b , 'breadotheranvilevade'

store
first
might
```

```
apl> c # -1 2 -2
apl> c U b
```

```
hater
sheet
there
metro
erase

bread
other
anvil
evade
```

```
mig
hat

met
era

apl> 0U'intact'

intact
```

```
apl> L # k # 3 2 4 R 'ABCDEFGH' , (I8) , 'abcdefgh'

 A B C D
 E F G H

 1 2 3 4
 5 6 7 8

 a b c d
 e f g h

apl> 0 2 U 3 5RI15

 3  4  5
 8  9 10
13 14 15

apl> Ry#3 1Uh#2 3R'abcdef'

0 2

apl> L # j # 0 U 4

4

apl> L # k # 0 0 0 U 4

4
```

```
apl> L # z # 2 1 3 U k

 h

apl> R z

1 1 1

apl> L # w # 5 U 23 41 73 26

.

apl> R w

0

apl> R m # 2 3 U h

0 0

apl> R j

1

apl> R k

1 1 1
```

```
apl> L#d#'a' 'an'('ant' 'ante')

 a an   ant ante

apl> `= d

3

apl> `= s

2

apl> R t

1

apl> L # u # 'abcde'

abcde
```

```
apl> R d

3

apl> L # s # -1 U d

 a an

apl> L # t # -2 U d

 a

apl> `= t

1

apl> (2 U u) # I 3
apl> u

 ab 1 2 3
```

```
apl> L # v # 3 4 R 'abcdefghijkl'

abcd
efgh
ijkl
```

Drop with Axis

For the CAPLIB^2 expression -

```
z # q U[x] v
```

these statements are all true:

- Subarrays are removed from the beginning or end of the **x[i]**th axis of **v**, according to whether **q[i]** is positive or negative.
- Axis did not apply to drop in classic APL.
- The shape along axes not selected by **x** are unchanged.
- All items in **x** must index an axis of **v**. This expression must produce a vector of 1s, or an error is indicated:

```
xEIRRv
```

- The axes in **x** need not be in ascending order, but no axis may be repeated.
- This CAPLIB^2 expression always produces 1:

```
&/,(qUv)=qU[IRRv]v
```

- The rank of the result is the same as the rank of **v**.

```
apl> L#a#3 4R'foldbeatrode'

fold
beat
rode
```

```
apl> 1U[1]a

beat
rode
```

```
apl> 1U[2]a

old
eat
ode
```

```
apl> 1 0Ua

beat
rode
```

```
apl> 0 1Ua

old
eat
ode
```

```
apl> L#q#3 2 4R'ABCDEFGH',(I8),'abcdefgh'

 A B C D
 E F G H

 1 2 3 4
 5 6 7 8

 a b c d
 e f g h
```

```
apl> 1 -1U[2 3]q

 E F G

 5 6 7

 e f g
```

```
apl> 1 -1U[3 2]q
```

```
 B C D

 2 3 4

 b c d

apl> L#t#'w' 'we'('wee' 'weed')'b' 'be'('bee' 'beep')

 w we   wee weed   b be   bee beep

apl> L#u#2 3Rt                    apl> ‘= u

 w we   wee weed                  3
 b be   bee beep

apl> L#q#1U[1]u                   apl> ‘= q

 b be   bee beep                  3

apl> Rq

1 3

apl> L#m#-1U[2]u                  apl> ‘= m

 w we                             2
 b be

apl> Rm

2 2

apl> L#n#-2U[2]u                  apl> ‘= n

 w                                1
 b

apl> Rn

2 1
```

Selective Specification

Drop can be used for selective specification, new with APL2, and also described on page 48:

```
apl> L # v # 3 4 R 'abcdefghijkl'

abcd
efgh
ijkl

apl> (1 -1 U v) # 2 3 R I 6
apl> v

 a b c d
 1 2 3 h
 4 5 6 l
```

```
apl> L#v#3 4R'abcdefghijkl'

abcd
efgh
ijkl
```

```
apl> (1U[1]v)#2 4RI8
apl> v

 a b c d
 1 2 3 4
 5 6 7 8
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Drop; Drop2; Endoper; Iscalar; Litvect;
     Quadout; Reshape;
     #include "aplinit.h"

     endoper(quadout(drop(iscalar(2),chrvect("abcde"))));
     endoper(quadout(drop2(
          iscalar(1), /* Number of items to drop. */
          reshape(litvect("3 4"),chrvect("foldbeatrode")),
          iscalar(2)))); /* Axis. */
}

cde
```

```
old
eat
ode
```

See Also

Ceiling (S), Minus (_), Reshape (R), Shape (R), Take (Y).

DYADIC

Dyadic is both the name of a CAPLIB^2 routine and an attribute of certain APL primitive functions and operators, which are said to be **dyadic** if they accept two arguments.

The CAPLIB^2 routine named **dyadic** extends dyadic scalar functions such as **plus** and **minus** to conformable pairs of APL variables containing double floating point data item-by-item. If the APL variables aren't double floating point, **dyadic** will do the necessary conversions.

The arguments must either agree in shape, or one must have only a single element, which is repeatedly matched with elements of the other. If both arguments have one item, but differing ranks, the result has the higher rank.

See Also

Numerous examples of the use of **dyadic** exist throughout this manual. See, for example, Plus (+), Minus (_), Divide (%). See also NILADIC, and MONADIC.

EACH ¨ ‘1

CAPLIB^2 operator only. **Each** accepts a function and either one or two arrays as arguments, depending on whether the function is dyadic or monadic: If dyadic, **each** applies the function to corresponding pairs of items of the two arrays; If monadic, the function is applied to each item of the array argument.

Dyadic Each

For any CAPLIB^2 expression of the form -

```
z # left function ‘1 right
```

where **function** is dyadic, these statements are all true:

- The shape and rank of **z** is the same as either **left** or **right**.
- Either **left** and **right** must have the same shape, or one is a scalar or vector of length one which is replicated by **each** to match the shape of the other.
- **Each** applies **function** to corresponding pairs of items from **left** and **right**. In other words, for any scalar **i** for which **pick (‘X)** is defined, these expressions produce identical results:

  ```
  i ‘X z
  (i ‘X left) function i ‘X right
  ```

- If either **left** or **right** is empty, the results of **each** in IBM's APL2 may not be the same as CAPLIB2, where the implementation is illustrated by this example:

  ```
  apl> a#3 0 2R0
  apl> b#1 2 3
  apl> " s is the shape of the empty argument.
  apl> L#s#,‘X(,‘X0=R‘1,‘1a b)/R‘1a b

  3 0 2

  apl> L#c#aR‘1b

  .

  apl> L#d#sR‘Z(Ya)RYb

  .

  apl> c ‘= d

  1
  ```

APL Usage

```
apl> L#z#4 6R‘1’me’ ’you’          apl> Rz
```

```
 meme youyou

apl> R ‘1 z

 4  6

apl> ’set’ , ‘1 ’hes’

 sh ee ts

apl> (‘Z 2 3) R ‘1 4 6

 4 4 4   6 6 6
 4 4 4   6 6 6

apl> Rz

0
```

```
2

apl> ‘=z

2

apl> 2 R ‘1 3 4 5

 3 3  4 4  5 5

apl> L#z#5Y‘1 0R‘Z0 0 0

.

apl> RYz

5
```

Monadic Each

For any CAPLIB2 expression of the form -

```
z # function ‘1 right
```

where **function** is monadic, these statements are all true:

- The shape and rank of the result **z** match the shape and rank of **right**.
- **Each** applies **function** to each item of **right**. In other words, for any scalar **i** for which **pick (‘X)** is defined, these expressions produce identical results:

  ```
  i ‘X z
  function i ‘X right
  ```

APL Usage

```
apl> L#z#R‘1’tom’ ’dick’

 3  4

apl> ‘=z
```

```
apl> Rz

2

apl> L#w#I‘1 1 2 3 4
```

```
2                                  1  1 2  1 2 3  1 2 3 4

apl> Rw                           apl> ‘= w

4                                 2

apl> display M ‘1 0 R ‘Z 2 3 R 0

+-------+
| +$--+ |
| U0 0| |
| |0 0| |
| |0 0| |
| +~--+ |
+E------+
```

C Usage

There is no direct call to **each**, but you can reach it from a **C** program with **compute**:

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Compute; Endoper; Quadout;
     #include "aplinit.h"

     endoper(quadout(compute(chrvect("I‘1 1 2 3 4"))));
}

 1  1 2  1 2 3  1 2 3 4
```

See Also

The **fill** function, on page 15

ENCLOSE ⊂ ‘Z

CAPLIB2 operator only. Creates a scalar whose only item is its argument. For any CAPLIB2 expression of the form -

```
z # ‘Z right
```

these statements are true:

- The rank of **z** (RRz) is 0.
- The shape of **z** (Rz) is the empty vector.
- If **right** is a simple scalar, it's the same as the result **z**.
- Disclose is the left inverse of enclose, i.e. $right \leftrightarrow `X`Zright$.

APL Usage

```
apl> L#a#2 3 4RI24

 1  2  3  4
 5  6  7  8
 9 10 11 12

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

apl> L#z#‘Za

  1  2  3  4
  5  6  7  8
  9 10 11 12

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

apl> L#d#'on' 'up' 'by'

 on up by

apl> Rd

3

apl> L#s#15 0 29
```

```
apl> Ra

2 3 4

apl> RRa

3

apl> ‘= a

1

apl> Rz

.

apl> RRz

0

apl> ‘= z

2

apl> ‘= d

2

apl> Rs
```

```
15 0 29                          3

apl> ‘=s

1

apl> s[2]#’none’

s[2]#’none’
^ 112 - bad number of replacement values.

apl> s[2]#‘Z’none’               apl> Rs

                                 3
apl> ‘=s

2

apl> L#t#3 5+‘Z0 1               apl> Rt

 3 4  5 6                        2

apl> L#z#(‘Z’st. ’),‘1q#’louis’ ’croix’

 st. louis st. croix


apl> L#w#,‘Z15 0 29              apl> Rw

 15 0 29                         1

apl> RRy#‘Z,5                    apl> ‘= y

0                                2
```

Enclose with Axis

For any CAPLIB^2 expression of the form -

`z # ‘Z[x] right`

these statements are true:

- **x** must be a simple scalar or vector.

- The set of axes indicated by **x** is enclosed.
- All the values in **x** must be proper axes, relative to the index origin (Lio) in force, i.e. if **x** is not empty, this statement must produce 1:

```
&/xEIRRright
```

- All of these statements are equivalent:

```
Rz
(Rright)[(IRRright)~x]RYz
(Rright)[,x]RRz
(RRright)_R,x
```

- If **x** is an empty vector, **z** is identical to **right** if it's a simple array. However, if **right** is nested, an empty **x** results in a copy of **right** with each item enclosed. In other words it produces the same as this expression:

```
‘Z‘1right
```

- The order of the axis in **x** determines the shape of each item of the result **z**.
- These two expressions produce identical results:

```
‘Zr
‘Z[IRRr]r
```

- Disclose with axis is the left inverse of enclose with axis. This expression produces a result identical to **right**:

```
‘X[x]‘Z[x]right
```

APL Usage of Enclose with Axis

```
apl> L#a#2 3RI6                    apl> L#z#‘Z[1]a

1 2 3                               1 4  2 5  3 6
4 5 6

apl> Rz                            apl> R‘1z
```

```
3

apl> ‘= z

2

apl> L#y#‘Z[2]a

 1 2 3  4 5 6

apl> Ry

2

apl> R‘1y

 3  3

apl> ‘=y

2

apl> L#w#‘Z[2]b

 pine odor data

apl> R‘1w

 4  4  4

apl> L#v#‘Z[I0]c#2 3RI6

1 2 3
4 5 6

apl> ‘=v

1

apl> L#q#2 3R’cat’ ’dog’ ’fox’ ’cow’ ’bat’ ’yak’

 cat dog fox
```

```
 2  2  2

apl> L#b#3 4R’pineodordata’

pine
odor
data

apl> L#x#‘Z[1]b

 pod ida not era

apl> Rx

4

apl> R‘1x

 3  3  3  3

apl> ‘=x

2

apl> Rw

3

apl> ‘=w

2

apl> Rv

2 3

apl> Rq
```

```
 cow bat yak

apl> ‘=q

2

apl> L#h#‘Z[IO]q

  cat   dog   fox
  cow   bat   yak

apl> ‘=h

3

apl> L#s#2 3 4R’lesssomenonemoremanymost’

less
some
none

more
many
most

apl> R‘1p

 3 4  3 4

apl> L#q#‘Z[3 2]s

 lsn  mmm
 eoo  oao
 smn  rns
 see  eyt

apl> ‘=q

2
```

```
2 3

apl> Rh

2 3

apl> L#p#‘Z[2 3]s

 less  more
 some  many
 none  most

apl> Rp

2

apl> ‘=p

2

apl> Rq

2

apl> R‘1q

 4 3  4 3
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Compute; Enclose; Endoper; Quadout; Scalar;
     #include "aplinit.h"

     endoper(quadout(enclose(
          compute(chrvect("2 3RI6")),
          scalar(1)
     )));
}
```

```
 1 4  2 5  3 6
```

See Also

disclose.

ENCODE ⊤ N

For **r** consisting solely of positive integers, **encode (r N x)**, is the inverse of **decode**. In other words these two statements produce identical results:

```
left B ( left N right )
( X / left ) | right
```

For a vector left argument and scalar right argument, **encode** is equivalent to this C routine:

```
encode(length,left,right,result)
int length; /* number of items in left */
double *left; /* vector of radices */
double right; /* value to encode */
double *result;
{
     double mod();
     int i;

     for (i = length; i != 0; i--) {
         *(result + i - 1) = mod(right,*(left + i - 1));
```

```
        if (*(left + i - 1) == 0)
            break;
        right = (right - *(result + i - 1))
          / *(left + i - 1);
    }
}
```

The above C program is equivalent to this APL defined function:

```
    G z#left encode right;i;Lct
[1]   Lct#0
[2]   z#0Xleft
[3]   i#Rleft
[4]   go:$(i=0)/0
[5]   z[i]#left[i]|right
[6]   $(l[i]=0)/0
[7]   right#(right_z[i])%left[i]
[8]   i#i_1
[9]   $go
    G
```

For any CAPLIB2 expression of the form -

```
z # left N right
```

these statements are true:

- The shape of the result is given by -

  ```
  (Rleft),Rright
  ```

- The rank of the result if given by -

  ```
  (RRleft)YRRright
  ```

- For **left** an array, each vector along its first axis is applied to each item of **right**, with the results being arrayed along the first axis of the result.

APL Usage

```
" encode undoes the work of the decode function
apl> r  #  10 10 10 10             apl> r N 1776
apl> r  B  1 7 7 6
```

```
1776

apl> 10 10 10 10 N 123

0 1 2 3

apl> 10 10 N 123

2 3

apl> 10 10 10 N 215 486 72 219 3

2 4 0 2 0
1 8 7 1 0
5 6 2 9 3

apl> (T2 3R10 10 10 8 8 8) N 123

1 1
2 7
3 3

apl> 2 2 2 2 2 N 15

0 1 1 1 1

apl> 2 2 2 N 15

1 1 1

apl> 0 24 60 60 N 162507

1 21 8 27

apl> 10 10 10 N 215 345 7

2 3 0
1 4 0
5 5 7
```

```
1 7 7 6

apl> 10 10 10 N 123

1 2 3

apl> 10 N 123

3

apl> 24 60 60 N 8592

2 23 12

apl> 10 B 2 2 2 2 N 15

1111

apl> 2 2 2 2 N 15

1 1 1 1

apl> 26 60 60 N 162507

19 8 27

apl> L # l # 4 2 R 8 2

8 2
8 2
8 2
8 2

apl> l N 15

0 1
0 1
1 1
7 1
```

```
apl> (( D 1 + 2 @ 135) R 2) N 135

1 0 0 0 0 1 1 1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Encode; Endoper; Iscalar; Litvect; Quadout;

     endoper(quadout(encode(
       litvect("10 10 10 10"),iscalar(123)))); /* 0 1 2 3 */
     endoper(quadout(encode(
       litvect("10 10 10"),iscalar(123)))); /* 1 2 3 */
     endoper(quadout(encode(
       litvect("10 10"),iscalar(123)))); /* 2 3 */
     endoper(quadout(encode(
       iscalar(10),iscalar(123)))); /* 3 */
     endoper(quadout(encode(
       litvect("2 2 2"),iscalar(13)))); /* 1 0 1 */
     endoper(quadout(encode(
       litvect("-2 -2 -2"),iscalar(13)))); /* -1 -1 -1 */
     endoper(quadout(encode(
       litvect("2 0 2"),iscalar(13)))); /* 0 6 1 */
     endoper(quadout(encode(
       litvect("2 2 2"),iscalar(-13)))); /* 0 1 1 */
     endoper(quadout(encode(
       litvect("-2 2 -2"),iscalar(13)))); /* 0 1 -1 */
}
```

See Also

Assign (#), Decode (B), Reshape (R), Shape (R), Transpose (T).

ENDOPER

C program usage only, **endoper** does termination processing for an apl variable. The sole argument to **endoper** is a pointer to an APLCB, described on page 29. If it is marked *temporary* (flag APLTEMP is on), **endoper** will free the storage associated with that APLCB.

Subroutines and functions described in this book, such as DYADIC, MONADIC, QUADOUT, etc., that accept pointers to APLCBs as arguments, will call **endoper** after they've completed their processing. Subroutines and functions that create APLCBs and return pointers to them, such as CHRVECT, CONVECT, and LITVECT, unless otherwise indicated in this book, will mark them temporary. Thus, these APLCBs will be freed automatically the first time they are passed as an argument to DYADIC, MONADIC, etc.

Numerous examples of using **endoper** exist throughout this book. See, for example, APLCOPY, APLMSG, ASSIGN, BINOMIAL, DYADIC, MONADIC.

See Also

GETCB, PERM, TEMP.

ENLIST ε E

CAPLIB2 and APL2 operator only, did not exist in classic APL. Enlist creates a simple vector consisting of all the simple scalars in its argument. Think of it as **ravel** extended to nested arrays. In fact, if the argument to enlist is simple the result is identical to **ravel**.

APL Usage

```
apl> a#(10 20 30) 'ab'
apl> b#Ea
apl> b

 10 20 30 ab

apl> Rb

5
```

```
apl> c#'ale' 'beer' 'stout'
apl> L#z#Ec

alebeerstout

apl> Rz

12

apl> ‘=z

1
```

```
apl> L#h#(2 2RI4)(2 2R(5 6(2 2R7 8 9 10)11))'abcd'
```

```
 1 2       5    6    abcd
 3 4    7  8   11
        9 10

apl> L#k#Eh                        apl> Rk

 1 2 3 4 5 6 7 8 9 10 11 abcd     15

apl> ‘=k

1
```

Enlist and Selective Specification

Enlist can be used for selective specification, also described on page 48:

```
apl> a#(10 20 30) 'ab'
apl> (Ea)#I5
apl> a

 1 2 3   4 5

apl> Ra

2

apl> (Ea)#'abcd'
apl> a

  abcd abcd abcd     abcd   abcd

apl> Ra

2

apl> (Ea)#1
apl> a

  1 1 1 1   1 1 1 1   1 1 1 1      1 1 1 1    1 1 1 1

apl> Ra
```

```
2

apl> (Ea)#I20
apl> a

 1 2 3 4  5 6 7 8  9 10 11 12    13 14 15 16  17 18 19 20

apl> Ra

2

apl> (Ea)#'x'
apl> a

 x x x x   x x x x   x x x x     x x x x   x x x x

apl> Ra

2
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Compute; Endoper; Enlist; Quadout; Scalar;
     #include "aplinit.h"

     endoper(quadout(enlist(
          compute(chrvect("'ale' 'bear' 'stout'"))
     )));
}

alebearstout
```

See Also

Depth, Ravel.

ENTRYS.H

Most of the sample C programs in this book start with two lines similar to the first two in the following example:

```
 1 #define INCLUDES APLCB+APLMAIN+STDIO
 2 #include "includes.h"
 3 main()
 4 {
 5      Aplmsg; Assign; Chrvect; Endoper; Execute; Litvect;
 6      #include "aplinit.h"
 7
 8      assign("left",  litvect("1.0 2.0 3.0"));
 9      assign("right", litvect("4.0 5.0 6.0"));
10      endoper(execute(chrvect("left + right")));
11 }
```

In addition to other things, “includes.h” generates a “#include entrys.h” statement, which contains macros for declarations of all of the entry points in the CAPLIB^2 subroutine library, over seven hundred in total. The macro names are the same as their corresponding entry name, but with the first character capitalized. So, for instance, the “Assign” on line 5 in the sample program above expands to -

```
Aplcb assign(char *, Aplcb)
```

This ensures you get the proper declarations for parameters and returned values, and allows you to declare only those routines used.

See Also

“includes.h” on page 178.

EQUAL = =

1 is returned if the difference of the absolute values of the left and right arguments is less than or equal to **comparison tolerance Lct**, and 0 is returned otherwise.

APL Usage

```
apl> 1 2 3 = 3 2 1
```

```
0 1 0
```

C Usage

Subroutines **dbleq, dbleqp, eq, eqp, eqx**, and **ieq** are related. They differ in the data types of arguments accepted, and the value returned. They all, however, implement the **logical comparison** operation.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
Dbleq; Dbleqp; Dyadic; Endoper; Eq; Eqp; Eqx;
     Idyadic; Ieq; Litvect; Quadout; Relative;
     double left[2],rite[2],ret[2];
     int ileft,irite,iret;
     #include "aplinit.h"

     endoper(quadout(dyadic(dbleq,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(relative(eq,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(idyadic(ieq,
          litvect("1 2 3"),
          litvect("3 2 1"))));
     *left = 2.1;
     *rite = 5.6;
     dbleqp(left,rite,ret);
     printf("dbleqp returned %f\n",*ret);
     eqp(left,rite,&iret);
     printf("eqp returned %d\n",iret);
     *left = 2.0; *(left + 1) = 0.0;
     *rite = 3.0; *(rite + 1) = 2.0;
     eqx(left,rite,&iret);
     printf("eqx returned %d\n",iret);
}
```

```
0 1 0
0 1 0
```

```
0 1 0
dbleqp returned 0.000000
eqp returned 0
eqx returned 0
```

See Also

IDBLEQ, IEQ, Comparison Tolerance (Lct).

)ERASE

)erase obj1 obj2 $\cdots$

The named objects (functions, variables, etc.) are expunged from the active workspace. If a name is a group name (see)GROUP), the group and any of its members are erased.

See Also

Expunge (Lex), Appendix E.

EXECUTE ⍎ C

Monadic **execute** accepts a character vector or scalar as its argument, evaluating the APL statement it represents. **Execute** can be invoked anywhere within an APL statement, but its result must be acceptable to its context. So **execute** applied to an empty vector, one that contains only spaces, starts with a right arrow (branch) or a comment symbol (") can appear only on the extreme left, as all these instances produce no explicit result.

APL Usage

```
apl> + / C '3 4 5'

    12

apl> 1+C'I4'
```

```
apl> C'I4'

1 2 3 4

apl> C'195%5X13'
```

```
2 3 4 5

apl> C'matrix#3 3RI9'
apl> matrix

1 2 3
4 5 6
7 8 9

apl> 2XC''

1

apl> Gf x
[1]       z#3Xx
[2]       G

apl> s#C'f 2'

s#C'f 2'
 ^ 70 - missing right argument during execute.
```

```
3

apl> C'''agnes'''

agnes

apl> v#I0
apl> C(0=Rv)/'''empty'''

empty

apl> ctr#0
apl> C(1=ctr)/'124'


apl> ctr#1
apl> C(1=ctr)/'124'

124
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Execute;
     #include "aplinit.h"

     endoper(execute(chrvect("L # 8 X 9")));
}

72
```

See Also

Reduction (/).

EXECUTE ALTERNATE ⎕EA **Lea**

For the CAPLIB2 statement **q Lea r**, where both **q** and **r** are simple character vectors or scalars, the interpreter first attempts to execute the expression in **r**. If that fails, the expression in **q** is attempted, without any message being generated for the failure in **r**. The result of the entire expression will be the result of the first expression to succeed.

APL Usage

```
apl> 'I3' Lea 'I4.5'             apl> 'I3' Lea 'I4'

1 2 3                            1 2 3 4

apl> 'I3.3' Lea 'I4.5'

'I3.3' Lea 'I4.5'
^ 45 - argument to indexgen must be a positive integer.

apl> '''err''' Lea 'I4.5'        apl> L#z#'3X2' Lea '$0'

err                              6

apl> Gdescribe

[1]       'the xyz workspace'
[2]       'provides several . . .'
[3]       G

apl> describe

the xyz workspace
provides several . . .

apl> L#z#'3X2' Lea 'describe'

the xyz workspace
provides several . . .
L#z#'3X2' Lea 'describe'
^ 94 - user function without result must stand alone.
```

```
apl> L#z#'describe' Lea '!-3'

the xyz workspace
provides several . . .
L#z#'describe' Lea '!-3'
^ 94 - user function without result must stand alone.
```

See Also

Atomic Vector (Lav), Index Of (I).

EXECUTE CONTROLLED □EC **Lec**

Quad ec is an APL2 system variable not implemented in CAPLIB^2.

EXPAND ⍀ \ '\ \

This CAPLIB^2 expression -

```
z # u \ x
```

expands **x** according to the pattern of ones and zeros in **u**, filling in with the fill item, given by -

```
YOR'ZYx
```

- The axis operator can be used with **expand**:

  ```
  z # u \[y] x
  ```

  If omitted the default axis is the last if \ is used, and the first if '\ is used.

- The shape of **x** must either equal the CAPLIB^2 expression **(+ / u)**, or the axis dimension must be one, or **x** must be a scalar, treated by **expand** as though it were a one-item vector. In the latter two cases, **x** is replicated along its axis dimension to the proper shape.

- These CAPLIB^2 statements all produce 1:

  ```
  (-1URz) '= -1URr
  (-1YRz) '= R,u
  (RRz) '= RRu
  ```

- Expand is like replicate with negative **u**. These statements produce 1:

```
(u\x) ‘= (u_u=0)/x
(u\x) ‘= (-1+2Xu)/x
```

- If **u** is empty, **x** must be a scalar or the shape along the axis dimension must be 0 or 1. If **x** is empty with a zero **y** axis, **u** must consist entirely of 0's. If **x** is empty with a nonzero **y** axis, **u** must contain the number of items in the **y** axis of **x**.

APL Usage

```
apl> L  #  x  #  3 4  R  'abcdefghijkl'

abcd
efgh
ijkl
```

```
apl> 1 0 1 1 1 \ x

a bcd
e fgh
i jkl
```

```
apl> 1 0 1 1 \ [1] x

abcd

efgh
ijkl
```

```
apl> 1 0 1 1 ‘\ x

abcd

efgh
ijkl
```

```
apl> 1 0 1 0 0 1 \ 1 2 3

1 0 2 0 0 3
```

```
apl> 1 0 1 0 0 1 \ 'abc'

a b  c
```

```
apl> L#h#(1 2) (3 4 5) 6

 1 2  3 4 5  6
```

```
apl> L#k#1 (2 3) (4 5 6)

 1  2 3  4 5 6
```

```
apl> 1 0 1 1 0 \ h

 1 2  0 0  3 4 5  6  0 0
```

```
apl> 1 0 1 1 0 \ k

 1 0  2 3  4 5 6  0
```

```
apl> r # 1 2 3 4 'a' 4 'c' 2 6
apl> r#r, 'x' 7 'y' 1 'd' 'e'
apl> L#r#5 4Rr,5 'f' 'g' 'h' 'i'
```

```
 1 2 3 4
 a 4 c 2
 6 x 7 y
 1 d e 5
 f g h i

apl> 1 0 0 1 1 0 1 \ r

 1 0 0 2 3 0 4
 a     4 c   2
 6 0 0 x 7 0 y
 1 0 0 d e 0 5
 f     g h   i


apl> 1 0 0 1 0 1 \ w # 7 8 9

7 0 0 8 0 9

apl> 1 -2 1 -1 1 / w

7 0 0 8 0 9

apl> L#b#1 0 1 \ 0 2R0

.

apl> Rb

0 3


apl> L#c#0 0 0 \ 2 0 R 0

0 0 0
0 0 0
```

```
apl> 1 0 0 \ 5

5 0 0

apl> L#s#3 1 R 7 8 9

7
8
9

apl> 0 1 0 \ s

0 7 0
0 8 0
0 9 0

apl> L#z#(I0)\2 0R0

.

apl> Rz

2 0

apl> L#a#(I0)\,[I0]6 7 8

.

apl> Ra

3 0

apl> Rc

2 3
```

Expand with Axis

```
apl> L#r#2 3 4RI24
```

```
apl> ((,r)[1 3 14 16])#'acde'
apl> r
```

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

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

apl> 1 1 0 1\[2]r

  a  2  c  4
  5  6  7  8

  9 10 11 12

 13  d 15  e
 17 18 19 20
  0  0  0  0
 21 22 23 24
```

```
  a  2  c  4
  5  6  7  8
  9 10 11 12

 13  d 15  e
 17 18 19 20
 21 22 23 24

apl> L#f#2 2 2R‘Z[2]8 2RI16

   1 2    3 4
   5 6    7 8

  9 10  11 12
 13 14  15 16

apl> 1 0 1\[2]f

   1 2    3 4
   0 0    0 0
   5 6    7 8

  9 10  11 12
   0 0    0 0
 13 14  15 16
```

```
apl> L#g#2 2 2R1 (2 3) 4 (5 6) 7 (8 9) 10 (11 12)

  1    2 3
  4    5 6

  7    8 9
 10   11 12

apl> 1 0 1\[2]g

  1    2 3
  0      0
  4    5 6

  7    8 9
```

```
apl> L#t#2 1 3RI6

1 2 3

4 5 6

apl> 1 0 0 1\[2]t

1 2 3
0 0 0
0 0 0
1 2 3
```

```
  0      0
 10   11 12
```

```
4 5 6
0 0 0
0 0 0
4 5 6
```

```
apl> L#m#3 4R'a' 'b' 1 'c' 2 3 4 5 6 7 8 9

 a b 1 c
 2 3 4 5
 6 7 8 9
```

```
apl> 1 0 0 1 1 ‘\ m

 a b 1 c


 2 3 4 5
 6 7 8 9
```

```
apl> 1 0 0 1 1 \[1] m

 a b 1 c


 2 3 4 5
 6 7 8 9
```

Expand and Selective Specification

Expand can be used for selective specification, also described on page 48:

```
apl> m#'abc'
apl> (1 0 1 0 1 \ m)#I5
apl> m

1 3 5

apl> L#t#2 4R'abcdefgh'

abcd
efgh

apl> L#m#3 2RI6

1 2
3 4
5 6
```

```
apl> L#n#2 3RI6

1 2 3
4 5 6


apl> (1 0 1 1\n)#t
apl> n

acd
egh

apl> (1 1 0 0 1\[1]m)#5 2R_I10
apl> m

-1  -2
-3  -4
-9 -10
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Expand; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(expand(
       litvect("1 0 1"),litvect("1 2"),1)));
}

1 0 2
```

See Also

Assign (#), Axis ([]), Compress (/), Quad Output (L), Reshape (R).

EXPONENTIAL ⋆ *

Returns the base of the natural logarithms to the power of the argument.

APL Usage

```
apl> * -2 -1 0 1 2

 .13533528 0.36787944 1 2.718282 7.389056

apl>*0j1

.5403023j.8414710

apl>*O0j1

-1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
```

```
{
     Endoper; Exp; Litvect; Monadic; Quadout;
     #include "aplinit.h"

     endoper(quadout(monadic(exp,
          litvect("0 1 2 3 4"))));
}
```

```
1 2.7182818 7.389056 20.085537 54.59815
```

See Also

Logarithm (`@`), POWER.

EXPUNGE ⎕EX Lex

Quad ex is a monadic system function that accepts either a character vector containing a single name, or a matrix containing a name in each row, of item(s) to be expunged from the workspace. Anything can be expunged except a label, group, or a halted function. **Quad ex** returns 1 if the name is now available for use, and 0 if it isn't.

APL Usage

```
apl> 'name "a" ',((4 X 0 ^= Lnc 'a') Y 'not'),'available'

name "a" available

apl> L # a # I 5

1 2 3 4 5

apl> 'name "a" ',((4 X 0 ^= Lnc 'a') Y 'not'),'available'

name "a" not available

apl> Lex 'a'

1

'name "a" ',((4 X 0 ^= Lnc 'a') Y 'not'),'available'
```

```
name "a" available
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Execqex; Execute; Quadout;
     #include "aplinit.h"

     avail("a");
     endoper(execute(chrvect("L # a # I5")));
     avail("a");
     endoper(quadout(execqex(chrvect("a"))));
     avail("a");
}

avail(name)
char *name;
{
     Chrvect; Execqnc; Ivalue;

     switch(ivalue(execqnc(chrvect(name)))) {
       case 0:
         printf("Name '%s' is available for use.\n",name);
         break;
       case 1:
         printf("Name '%s' is a label name.\n",name);
         break;
       case 2:
         printf("Name '%s' is a variable name.\n",name);
         break;
       case 3:
         printf("Name '%s' is a function.\n",name);
         break;
     }
}
```

```
Name 'a' is available for use.
1 2 3 4 5
```

```
Name 'a' is a variable name.
1
Name 'a' is available for use.
```

See Also

Assign (#), Catenate (,),)Erase, Name Classification (Lnc), Not Equal ($\wedge =$), Take (Y), Times (X).

FACTORIAL ! !

For positive integers `( ! n)` produces the product of all positive integers up through `n`. !0 produces 1. Trying to apply factorial to a negative integer is an error. For any other argument, the result is obtained from the gamma function:

$$\Gamma(n+1) = \int_0^\infty z^n e^{-x} dx$$

APL Usage

```
! 0 1 2 3 4

    1 1 2 6 24

! -.5 0 .5 1 1.5 2 2.5

    1.7724538509 1 0.8862269267 1 1.3293403882 2 3.3233509704
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
Endoper; Factorl; Factorlp; Factorlx;
     Ifactorl; Ifactorp; Imonadic; Litvect;
     Monadic; Quadout;
     double rite[2],ret[2];
     int irite,iret;
     #include "aplinit.h"
```

```
     endoper(quadout(monadic(factorl,
       litvect("0 1 2 3 4 -0.5 0.5 1.5"))));
     endoper(quadout(imonadic(ifactorl,
       litvect("0 1 2 3 4"))));
     *rite = -0.5;
     factorlp(rite,ret);
     printf("factorlp returned %f\n",*ret);
     *rite = 4.0;
     *(rite+1) = 2.0;
     factorlx(rite,ret);
     printf("factorlx returned %fj%f\n",*ret,*(ret+1));
irite=5;
     ifactorp(&irite,&iret);
     printf("ifactorp returned %d\n",iret);
}
```

```
1 1 2 6 24 1.7724539 0.8862269 1.3293404
1 1 2 6 24
factorlp returned 1.772454
factorlx returned -3.011540j1.770168
ifactorp returned 120
```

FIND ε ‘E

CAPLIB2 and APL2 operator only, did not exist in classic APL. The expression -

```
z # left ‘E right
```

yields a boolean array **z** of 1's and 0's, of shape and rank identical to **right**. An item of **z** is 1 wherever the pattern given by **left** begins in the corresponding position of **right**.

If **left** is smaller of rank, the search is along the last **RRleft** axes of **right**, with **left** treated as though it were reshaped by the expression -

```
(((RRright)_RRleft)R1),Rleft)Rleft
```

If **left** is larger of rank, z will contain all 0's.

APL Usage

```
apl> 'ab'‘E 'ababababa'          apl> L#q#2 3R'abcabb'
```

```
1 0 1 0 1 0 1 0 0

apl> L#h#4 5R'abcaba'

abcab
aabca
baabc
abaab

apl> L#k#2 3R'bcaabc'

bca
abc

apl> k `E h

0 1 0 0 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0

apl> 'ba' `E h

0 0 0 0 0
0 0 0 0 0
1 0 0 0 0
0 1 0 0 0
```

```
abc
abb

apl> q `E 'abcabb'

0 0 0 0 0 0

apl> s#'go' 'on' 'go' 'to'
apl> 'go' 'to' `E s

0 0 1 0

apl> 'goto' `E s

0 0 0 0

apl> (~'  '`Es)/s#'ab    def'

ab def
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Find; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(find(
       litvect("1 2 3"),
       litvect("1 2 3 4 1 2 3")
     )));
```

```
}
```

```
1 0 0 0 1 0 0
```

See Also

Comparison Tolerance (Lct).

FIRST ↑ Y

CAPLIB2 and APL2 operator only, did not exist in classic APL. The expression -

```
z # Y right
```

yields the first item of **right**, unless it's empty, in which case it yields the prototype of **right**.

APL Usage

```
apl> L#a#'do' 're' 'me'

 do re me

apl> ‘= z

1

apl> y#'abcde'
apl> ‘= y

1


apl> ‘=b#(2 3) ((4 5 6) 7)

3


apl> L#c#‘Ob
```

```
apl> L#z#Ya

do

apl> Rz

2

apl> L#w#Yy

a

apl> ‘=w

1

apl> L#j#Yb

2 3

apl> ‘=j
```

```
  4 5 6  7   2 3

apl> L#s#Yc

 4 5 6  7

apl> ‘=c[1]

3

apl> Rd#0 3R(2 3R0) 0

0 3

apl> Yd

0 0 0
0 0 0

apl> Rh#0 2R(0 0) (0 0 0)

0 2
```

```
1

apl> Rs

2

apl> YI0

0

apl> Rd

0 3

apl> ' '=Y''

1

apl> Yh

0 0
```

First and Selective Specification

First can be used for selective specification, also described on page 48:

```
apl> L#k#'red' 'white' 'blue'

 red white blue

apl> (Yk)#'yellow'
apl> k

 yellow white blue
```

```
apl> Rk

3

apl> ‘=k

2
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; First; Quadout;
```

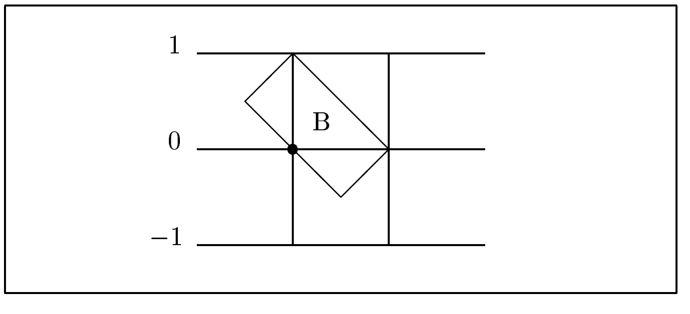
```
     #include "aplinit.h"

     endoper(quadout(first(chrvect("abcde"))));
}

a
```

See Also

Pick, Enclose.

FLOOR ⌊ D

Downstile (D) when used monadically yields the largest integer not greater than its argument for real numbers. The figure below represents the complex number plane, the horizontal axis being of course the real part, and the vertical the imaginary part. Point **B** is the **floor** for any complex number falling in the slanted rectangle, or on one of its left edges. The rectangle has dimensions *square root of* 2 *by square root of* .5 ($1.414 \times .707$):



APL Usage

```
apl> D -3.14  2.718              apl> D 2.3 -2.7 3 .5

-4 2                             2 -3 3 0

apl> D 1.5j2.5 1j2 1.2j2.5 -1.2j-2.5

2j2 1j2 1j2 -1j-3
```

C Usage

There is no **floor** routine in CAP2.LIB. Instead, CAPLIB2 relies on the ANSI Standard **floor** routine being available.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Floor; Floorp; Floorx;
     Litvect; Monadic; Quadout;
     double num[2],ret[2];
     #include "aplinit.h"

     endoper(quadout(monadic(floor,
          litvect("-3.14 2.718"))));
     *num = -3.14;
     floorp(num,ret);
     printf("floorp returned %f\n",*ret);
     *num = 1.5;
     *(num+1) = 2.5;
     floorx(num,ret);
     printf("floorx returned %fj%f\n",*ret,*(ret+1));
}

-4 2
floorp returned -4.000000
floorx returned 2.000000j2.000000
```

See Also

Ceiling (S).

)FNS

Reports in alphabetical order, starting with the indicated letter if given, a list of the functions in the active workspace. IBM's APL2 permits a range of starting letters, as in ")FNS I-P". This is not implemented in CAPLIB2.

APL Usage

)fns [**a**]

See Also

Name List (Lnl), Appendix E

FORMAT ⍕ F

Format can be invoked both monadically and dyadically. All forms return a character array containing a formatted representation of the right argument suitable for display.

The monadic form of **format** is implicitly invoked any time you either assign an APL expression to **quad (L)**, as in **(a # L # 1 3 5)**, or the last operator invoked in an expression is other than assignment, as in **(1 + 2)**.

The dyadic form of **format** has two flavors, **Format by Example**, and **Format by Specification**, each treated separately below.

Format, Monadic Form

For the CAPLIB2 expression -

```
z # F right
```

these statements are true:

- The result **z** is a character array with representations of all data in **right**, along with leading blanks and blank lines to indicate levels of nesting, and dimensionsal relationships.
- If **right** is a simple character array, **z** will be identical to it.
- If **right** is simple, the rank of **z** is given by **,1YRRright**. If **right** is nested, **z** will have rank either 1 or 2. If all items of **right** at any depth are scalars or vectors, **z** is of rank 1. Otherwise, **z** is a matrix.
- For nested arrays, there is one column each of leading and trailing blanks for each level of nesting.

```
apl> L#r#2 3R'one' 1 1 'two' 2 22

 one 1  1
 two 2 22
```

```
apl> L#n#Fm#'3  5  six'
```

```
apl> L#z#Fr

 one 1  1
 two 2 22

apl> Rz

2 10

apl> L#c#Fb#(2 3 4) 5 (‘Z7 8 (9 10 11))

 2 3 4  5   7 8  9 10 11

apl> Rb

3

apl> ‘=b

4

apl> Rc

27

apl> b‘=c

0

apl> L#d#(1 2) (3 4 5) 6 (2 2 3R6+I12)

 1 2  3 4 5  6     7  8  9
                  10 11 12

                  13 14 15
                  16 17 18

apl> RL#e#Fd

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

```
3  5  six

apl> n‘=m

1

apl> L#y#Fs#4XI4

4 8 12 16

apl> s‘=y

0

apl> Rs

4

apl> Ry

9
```

```
        13 14 15
        16 17 18

5 27
```

Format by Specification

In classic APL, **Format by Specification** accepted only a numerical array as its right argument. The left argument was (and still is in APL2 and CAPLIB2) used to control the format of the result, and could be either a single number, a pair or numbers, or a vector of length **(2 X -1 Y 1 , R a)**, "**a**" being the right argument.

If the left argument to **format** isn't a vector of the desired length, it will be expanded as needed. First, if it is a single number, it is treated as though it were the second of a pair of numbers, the first number being an implied zero. A pair of numbers (explicit or implicit) will be treated as though it had been repeated sufficient times to produce a vector twice the length of the last dimension of the right argument.

The left argument to **format** is considered to be a vector of paired control numbers, each pair consisting of a *width indicator* and a *precision* specification. Each pair controls the formatting of numbers with a particular index value in the last dimension of the right argument.

If the width indicator is zero, a width is derived so that at least one space will be left between adjacent columns of formatted numbers in the final result.

The sign of each precision indicator indicates the form to be used, negative indicating scaled form, and positive indicating decimal form. Positive precision indicators indicate the number of digits to the right of the decimal point in decimal form, and negative precision indicators indicate the number of digits in the multiplier of scaled form.

A number in decimal form is 123.45. The same number in scaled form, also called *scientific notation*, is $1.2345e+02$. If a number has more decimal positions than allowed for the a precision indicator, it is rounded. For example with a precision indicator of 2, .125 would print as .13, and .124 as .12.

All columns of the output are right-justified, unless an entire column contains only character data, in which case it's left justified. The numbers are also aligned on the decimal point. The precision indicators in the left argument apply of course only to the numbers.

The right argument to **Format by Specification** must have a depth no greater than 2. Further, each item of the right argument must be a simple numeric scalar or a simple character scalar or vector. The **each** operator discussed on page 103 can be used to recursively extend **Format by Specification** to nested arrays.

APL Usage, Format by Specification

```
apl> 4 1 F p # 2 3 5 7

2.0 3.0 5.0 7.0
```

```
apl> 8 -1 F p

2e+00 3e+00 5e+00 7e+00
```

```
apl> L # a # 3 2 R 12.34 -34.567 0 12 -0.26 -123.45

12.34  -34.567
 0.00   12.000
-0.26 -123.450
```

```
apl> 12 3 F a

      12.340     -34.567
       0.000      12.000
      -0.260    -123.450
```

```
apl> -2 F a

 1.2e+01 -3.5e+01
 0.0e+00  1.2e+01
-2.6e-01 -1.2e+02
```

```
apl> 6 2 12 -3 F a

  12.34   -3.46e+01
   0.00    1.20e+01
  -0.26   -1.23e+02
```

```
apl> 4 2 12 -5 F r # 3 2 R 1 .468987 2 57.276 3 27963

1.00   4.6899e-1
2.00    5.7276e1
3.00    2.7963e4

apl> 4 0 10 2 F r

   1      0.47
   2     57.28
   3  27963.00
```

```
apl> 7 2 F s # 3 2 R I 6

   1.00   2.00
   3.00   4.00
   5.00   6.00

apl> 3 F s

 1.000 2.000
 3.000 4.000
 5.000 6.000
```

```
apl> 15 0 F 2*70 " Should cause overflow

15 0 F 2*70 " Should cause overflow
     ^ 29 - a width is too small in format.

apl> Lfc[4]#'?'
apl> 15 0 F 2*70 " Should cause overflow

???????????????

apl> a#4 2R'amt' 'percent' 5 26.31 6 31.5 8 42.11
apl> 3 0 9 2 F a

amt  percent
  5    26.31
  6    31.50
  8    42.11

apl> 0 F a

 amt percent
   5      26
   6      32
   8      42

apl> L#d#'item' 'pens' 'books' 'paper',a

 item  amt percent
 pens    5   26.31
 books   6    31.5
 paper   8   42.11

apl> 5 0 5 0 9 2 F d " 1st col. s/b left justified, all char.

item   amt  percent
pens     5    26.31
books    6    31.50
paper    8    42.11

apl> 2 F 1 (2 3) " Should cause domain error.

2 F 1 (2 3) " Should cause domain error.
```

```
  ^ 133 - Domain error.

apl> 2 F ‘1 3 2 R(1 2) (3 4 5) 6 7 (8 9) 10

  1.00 2.00  3.00 4.00 5.00
  6.00       7.00
  8.00 9.00  10.00
```

Format by Example

CAPLIB2 and APL2 only, does not exist in classic APL. For the CAPLIB2 expression -

```
z # left F right
```

these statements are true:

The result **z** is a character array with representations of all data in **right**, along with leading blanks and blank lines to indicate levels of nesting, and dimensionsal relationships.

The right argument contains the data to be formatted, and must be a simple real numeric array.

The left argument must be a simple character vector that contains a model, or field, for each column of the result **z**. The "column" of **z** referred to here is different from what I normally think of as a column of an APL array, namely, each item distinctly indexed along the last dimension. For example, given -

```
apl> L#right#3 2R3 4.99 7 7.45 12 .5

 3 4.99
 7 7.45
12 0.50
```

to my thinking, the second column of **right** is given by:

```
apl> right[;2]

4.99 7.45 0.50
```

In the context of **format by example**, however -

```
apl> left#' 55@   $55.50 EA'
apl> L#z#left F right

  3@   $ 4.99 EA
```

```
  7@   $ 7.45 EA
 12@   $  .50 EA
```

the second "column" of **z** is given by:

```
apl> z[;7+I9]

$ 4.99 EA
$ 7.45 EA
$  .50 EA
```

There must be either one **field** or **model** in the left argument, which is applied repeatedly to each column of the right argument, or there must be as many **fields** as the right argument has columns (my sort of column, see above). In other words, if **left** has **-1YRright** fields, then **Rleft** will match **-1YRz**. But if **left** has one **field**, **-1YRz** will match **(Rleft)X-1YRright**.

A **field** or **model** in the left argument is a sequence of characters containing at least one digit and bounded by either blanks or a special field boundary mark (the digit 6). Spaces are retained in the result.

A blank-delimited sequence of characters not containing a digit is considered a "decoration." In the previous example, @, $, and "EA" were decorators. The spaces themselves are considered decorations.

Digits (1234567890), the period (.), and comma (,) are considered **"control characters"** when they appear in a **field**. They specify where numbers from **right**, decimals, controlled commas (thousands indicators, see below), and floating decorators (see below) appear in **z**.

Following are explanations and examples of all the field control characters. Note that the commonest to appear are the 5, 0, and the comma and period:

- '0' - Pad with 0's to the position of the 0. If the value of the corresponding item of r is 0, the position is filled with 0's:

  ```
  apl> ' 055.50' F .3 33.2 0 300

   000.30 033.20 000.00 300.00
  ```

- '1' - Float the decorator against the number only if the value is negative:

  ```
  apl> ' _55.10' F r # -3.4 0 4.5 -2.12

     _3.40    .00   4.50  _2.12
  ```

```
apl> ' (55.10)' F r

  (3.40)    .00    4.50   (2.12)
```

- '2' = Float the decorator against the number only if the value is positive:

```
apl> ' +552.50' F r # -4 40 -400

    4.00  +40.00  400.00

apl> ' -551.20cr' F r

   -4.00     40.00cr -400.00
```

- '3' = Float the decorator against the number.

```
apl> ' $555.50' F r # 3.1 32.23 324

 $  3.10 $ 32.23 $324.00

apl> ' $553.50' F r

   $3.10  $32.23 $324.00
```

- '4' = Counteract the effect of a 1, 2, or 3, to prevent it from affecting the other side of the decimal. Any decorator on the same side of the decimal as the 4 displays as entered:

```
apl> ' -551.20cr' F r # -1 10 -100

   -1.00     10.00cr -100.00

apl> ' -551.40cr' F r

   -1.00cr   10.00   -100.00cr
```

- '5' = perform normal formatting, removing leading and trailing 0's. Display blanks for a value of 0L

```
apl> '555_555_5555' F 8324632190

832 463 2190
```

- '6' = The decorator to the right marks the end of this field; treat it as though there were a blank between the fields, but display the decorator:

```
apl> '0006/06/06 06:06' F 1995 8 29 21 58

1995/08/29 21:58
```

- '7' = The next nonnumeric character to the right is the "e-symbol" in scientific notation:

```
apl> '  -1.7000E-01' F -25.784 .0034 123.45 -123.45

  -2.5784E 01   3.4000E-03   1.2345E 02  -1.2345E 02

apl> ' -15.7000E-01' F 123.45 -123.45

  12.3450E 01 -12.3450E 01
```

- '8' = Fill empty portions of the field with Lfc[3], for which the default is '*' (See also Format Control, on page 149):

```
apl> '  85555.50' F 17.3 56.43

  ***17.30  ***56.43
```

- '9' = Pad with 0's to the position of the 9. If the corresponding item of **right** is 0, the position is all blanks:

```
apl> '  9995.59' F 14.7 0 56.43

  0014.70           0056.43

apl> '  9995.59-' F 14.7 0 56.43

  0014.70-            0056.43-
```

More Examples of Format by Example

```
apl> l#' 55@  $55.50 EA'
apl> L#r#3 2R3 4.99 7 7.45 12 .5
```

```
 3 4.99
 7 7.45
12 0.50

apl> L#z#lFr

  3@  $ 4.99 EA
  7@  $ 7.45 EA
 12@  $  .50 EA

apl> Rz

3 15

apl> Rl

15

apl> expr#234.67 456.23 987.65 34.23
apl> 'total order cost:  $5,555.50'F+/expr

total order cost:  $1,712.78

apl> ' 55.55' F 12.345 12.344

 12.35 12.34

apl> ' 55.55' F .10 1.1 1.01 10.019 .11

   .1   1.1   1.01 10.02   .11

apl> ' 55.55' F 2 2.2 0 2.22

  2     2.2         2.22

apl< left#' 55.55 -55.10 x (55.10) +552.50'
apl> left#left,' -551.20CR x 0006/06/06 06:06 555-'
apl> left F I11
(Note: Some spaces were removed to make this fit on the page)

 1   2.00 x   3.00    +4.00    5.00CR x 0006/07/08 09:10  11-
```

```
apl> ' -551.20CR' F 5 -5

    5.00CR   -5.00
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Circular; Dyadic; Endoper; Form; Litvect;
     Aplcb answer,left=NULL;
     #include "aplinit.h"

     answer = form(left,dyadic(circular,
       litvect("-2.0"), /* inverse cosine */
       litvect("0.707")));
     printf(
       "The arc whose cosine is 0.707 is %s radians.\n",
       answer -> aplptr . aplchar);
     endoper(answer);
     answer = form(litvect("5 3"),dyadic(circular,
       litvect("-2.0"), /* inverse cosine */
       litvect("0.707")));
     printf(
       "The arc whose cosine is 0.707 is %s radians.\n",
       answer -> aplptr . aplchar);
     endoper(answer);
}

The arc whose cosine is 0.707 is  .7855492 radians.
The arc whose cosine is 0.707 is  .786 radians.
```

See Also

Assign (#), Catenate (,), Each(‘1), Format Control (Lfc), Reshape(R), Shape (R), Take (Y), Times (X).

FORMAT CONTROL ⎕FC **Lfc**

CAPLIB2 and APL2 only. Not defined in classic APL. Lfc is a vector of at least length five (any beyond the first five are ignored) containing characters used by **Format by Specification** (page 141) and by **Format by Example** (page 144).

- Lfc[1] - Decimal point, default '.'. Prints where a '.' is specified in the left argument to Format by Example.

```
apl> Lfc[1]#','
apl> 5 2 F 23.45

23,45

apl> 7 -2 F 2.45e15

 2,5e15
```

```
apl> Lfc[1]#'.'
apl> 5 2 F 23.45

23.45

apl> 7 -2 F 2.45e15

 2.5e15
```

- Lfc[2] - Thousands indicator, default ','. Prints where a ',' is specified in the left argument to Format by Example.

```
apl> total#+/234.67 456.23 987.65 34.23
apl> Lfc[2]#'&'
apl> 'total order cost:  $5,555.50' F total

total order cost:  $1&712.78

apl> Lfc[2]#','
apl> 'total order cost:  $5,555.50' F total

total order cost:  $1,712.78
```

- Lfc[3] - Blank fill for 8 in Format by Example, default '*'.

```
apl> '  85555.50' F 17.3 56.43

  ***17.30  ***56.43

apl> Lfc[3]#'%'
apl> '  85555.50' F 17.3 56.43

  %%%17.30  %%%56.43

apl> ' -85555.10' F -17.3 56.43

 %%%-17.30 %%%%56.43
```

- Lfc[4] - Fill for overflows, default '0'. If Lfc[4] is '0' an error is generated if the field is too small for the corresponding value in Format by Example. If Lfc[4] is not '0' an overflow causes the field to be filled with that character.

```
apl> Lfc[4]#'?'
apl> ' 55.55'F345 .6789

 ?????   .68

apl> Lfc[4]#'0'
apl> ' 55.55'F345 .6789

' 55.55'F345 .6789
        ^ 29 - a width is too small in format.
```

- Lfc[5] - Print-as-blank, default '_'.

```
apl> '555_555_5555' F 8324632190

832 463 2190

apl> Lfc[5]#'^'
apl> '555_555_5555' F 8324632190

832_463_2190

apl> '555^555^5555' F 8324632190

832 463 2190
```

- Lfc[6] - Negative number indicator, default '-'. IBM's APL2 recognizes a sixth character, Lfc[6], that prints as the sign of a negative number. This is not implemented in CAPLIB2, although you can get a similar effect with Format by Example:

```
apl> ' _55.10' F r # -3.4 0 4.5 -2.12

  _3.40    .00   4.50  _2.12
```

FUNCODES.H

Structures defined in funstruc.h are initialized describing all of the builtin

APL functions. These codes are stored in the “codes” field of each structure to identify its type:

```
#define MONADIC    1
#define DYADIC     2
#define SCALAR     4
#define MIXED      8
#define AXIS      16
#define EQUALITY  32 /* Equal/not equal is special case. */
#define DUAL      64 /* Function is both monadic, dyadic.*/
#define DBL      128 /* Double numeric only, axis value. */
#define FIRST    256 /* Default axis is first dimension. */
#define LAST     512 /* Default axis is last dimension.  */
#define OPER    1024 /* Operator.                        */
#define AXED    2048 /* Axes, may be vector, dyadic.     */
#define AXEM    4096 /* Axes, may be vector, monadic.    */
#define SESP    8192 /* Selective specification.         */
#define AXES AXED+AXEM             /* 2048 + 4096 = 6144 */
#define SCDO SCALAR+DYADIC         /* 4 + 2       =    6 */
#define SCMD SCALAR+MONADIC+DYADIC /* 4 + 1 + 2   =    7 */
#define MXMO MIXED+MONADIC         /* 8 + 1       =    9 */
  /* mixed, monadic only (compute)                       */
#define DYAE MIXED+DYADIC          /* 8 + 2       =   10 */
  /* find                                                */
#define MIXD MIXED+MONADIC+DYADIC  /* 8 + 1 + 2   =   11 */
#define UPAR MIXD+AXED             /* 11 + 2048   = 2059 */
#define MSMD SCMD+MIXED            /* 7 + 8       =   15 */
  /* monadic scalar, dyadic mixed, roll/deal not/without */
#define MIXA MIXD+AXIS             /* 11 + 16     =   27 */
#define EQNE SCDO+EQUALITY         /* 6 + 32      =   38 */
#define SAME MIXED+MONADIC+DYADIC+DUAL /* 8+1+2+64=   75 */
#define DYDB MIXED+DYADIC+DBL      /* 8 + 2 + 128 =  138 */
  /* decode, encode                                      */
#define MIXC MIXA+DBL              /* 27 + 128    =  155 */
  /* axis for laminate is double                         */
#define MIXF MIXA+FIRST            /* 27 + 256    =  283 */
#define MIXL MIXA+LAST             /* 27 + 512    =  539 */
#define OPAX OPER+AXIS             /* 1024 + 16   = 1040 */
  /* compress(slash) expand(slope)                       */
#define OPAF OPAX+FIRST            /* 1040 + 256  = 1296 */
  /* slash-bar/slope-bar                                 */
#define OPAL OPAX+LAST             /* 1040 + 512  = 1552 */
```

```
  /* slash/slope                                          */
#define DIXE DYAE+AXES            /* 10 + 6144   = 6154 */
  /* drop2                                                */
#define MIXE MIXD+AXES            /* 11 + 6144   = 6155 */
  /* enclose, partition, ravel2                           */
#define MIXG MIXD+AXEM            /* 11 + 4096   = 4107 */
  /* disclose, pick                                       */
#define DIXS DIXE+SESP            /* 6154 + 8192 =14346 */
  /* squadix                                              */
```

FUNCTION EDITING ▽ G

The del editor lets you edit defined functions. You can add new statements or display, delete, replace, or alter existing ones, one line at a time. It was designed long before present-day display consoles became commonplace. It works great if you happen to be working on a teletype-style hardcopy terminal.

Entering Del Editor

You enter the del editor by keying a *del* (`G`) followed by a function name.

Leaving Del Editor

You leave the del editor by keying a *del* by itself.

Adding or replacing a Statement

The editor normally prompts you with the bracketed number of the statement it expects you to enter next. If you don't precede it with a bracketed statement number of your own, the statement you enter is assigned the number in the prompt. If you enter just a bracketed number, the editor will prompt you with the same bracketed number, and the statement you then enter replaces one with a matching statement number, or is added at the appropriate location if there is no matching statement number existent.

Inserting a New Statement

If you wanted to insert a new statement between existing statements numbered 1 and 2, you can do so by prefixing yours with the proper fractional number, e.g. -

```
[0.1] a + b
```

Deleting a Statement

Delete statement 5, for example, by entering -

```
[H5]
```

Editing a Statement

Enter `[xLy]` to edit statement **x** according to these rules:

- Statement **x** is displayed with the cursor on the line immediately below it under position **y**, unless **y** is zero, in which case the cursor is positioned to the right of the rightmost character in the line. You can type whatever you want on this line, but everything is ignored except alphabetics, numbers, and a slash (/). A slash means delete the character in that position of the statement. A digit **z** means insert **z** spaces at that position in the statement. A character **a** thru **z** means insert $k \times 5$ spaces at that position, where **k** is the position of the character in the alphabet. For example, **a** means 5 spaces, **b** means 10, and so on. When you're done press **Enter**.

- The statement is re-displayed with the indicated characters deleted, and blanks inserted. You can replace anything in the line you want, or delete any character with the **Delete** key. I recommend you try appending a mismatched left parenthesis to the line. The editor gives you an error message, and places you in a special edit mode on the line which is not standard APL. It's more like the normal DOS environment, where **Backspace** deletes a character and shifts the cursor left, **Delete** deletes a character without moving the cursor, and the arrow keys move the cursor to different positions without changing anything.

Displaying Statements

While in the del editor, you can display the entire edited function by keying `[L]`. Lines from statement **n** onward are displayed if you enter `[nL]`. If you're not presently in the del editor and you'd like to display a defined function in del edit form without entering edit mode, you can do so by entering `Gname[L]G`, where **name** is the function name.

See Also

Appendix K, Canonical Representation (Lcr), Defined Function, Function Establishment (Lfx), Quad Output (L).

FUNSTRUC.H

Funstruc.h and funstrud.h define all the structures used to describe builtin APL functions and operators.

```
#if !defined(SCALAR_PROC_DEF)
#define SCALAR_PROC_DEF
typedef void (*SCALAR_PROC)(); /* Entry dyadic scalar. */
#endif
#include "funstrud.h"
typedef struct oper { /* Slash, slope, each, dot, etc. */
     Codes codes;
     int (*eptok)(Apltoken); /* Transform to derived. */
} Oper;
typedef struct dmxo { /* Dyadic mixed only, or dual. */
     Codes codes;
     Mixed_dyadics dyad;
} Dmxo;
typedef struct mixa { /* Mixed functions. */
     Codes codes;
     Mixed_monadics mona;
     Mixed_dyadics  dyad;
} Mixa;
typedef struct msmd {
/*monadic scalar mixed dyadic roll/deal not/without*/
     Codes codes;
     Scalar_monadics mona;
     Mixed_dyadics   dyad;
} Msmd;
typedef struct rela { /* relational functions */
     Codes codes;
     Scalar_dyadics dyad;
} Rela;
typedef struct eqne { /* equality functions */
     Rela rel;
     int    (*ch)(char, char);
     void   (*ph)(char*, char*, int*);
```

```
     void   (*pd)(char*, char*, double*);
     void   (*eqnedum)(void*, void*, int*);
} Eqne;
typedef struct scalars {
     Codes codes;
     Scalar_dyadics  dyad;
     Scalar_monadics mona;
} Scalars;
```

FUNSTRUD.H

These structures are used to describe all the builtin APL functions. Funstrud is INCLUDEd by funstruc.h.

```
typedef struct codes {
  int funky_code; /* Code, e.g. QUOTE_DOT. */
  int funky_flags; /* SCALAR, etc. - see funcodes.h. */
} Codes;
typedef struct identities {
     int iid; double did;
} Identities;
typedef struct scalar_dyadic_procs {
     void   (*ppint)(int*,int*,int*);
     void   (*ppdbl)(double*,double*,double*);
     void   (*ppmix)(double*,double*,int*);
     void   (*ppchr)(char*,char*,int*);
     void   (*ppcpx)(double*,double*,double*);
     void   (*ppmpx)(double*,double*,int*);
} Scalar_dyadic_procs;
typedef struct scalar_dyadic_funcs {
     int    (*epint)(int,int);
     double (*epdbl)(double,double);
     int    (*epmix)(double,double);
} Scalar_dyadic_funcs;
typedef struct scalar_dyadics {
     Identities identities;
     Scalar_dyadic_procs procs;
     Scalar_dyadic_funcs funcs;
} Scalar_dyadics;
typedef struct scalar_monadic_funcs {
     int    (*epint)(int);
```

```
     double (*epdbl)(double);
     int    (*epmix)(double);
} Scalar_monadic_funcs;
typedef struct scalar_monadic_procs {
     void   (*ppint)(int*,int*);
     void   (*ppdbl)(double*,double*);
     void   (*ppmix)(double*,int*);
     void   (*ppcpx)(double*,double*);
} Scalar_monadic_procs;
typedef struct scalar_monadics {
     Scalar_monadic_procs procs;
     Scalar_monadic_funcs funcs;
} Scalar_monadics;
typedef union mixed_monadics {
     Aplcb (*epmix)(Aplcb);
     Aplcb (*epmixa)(Aplcb, int); /* with axis */
     Aplcb (*epmixe)(Aplcb, Aplcb); /* with axes */
     Aplcb (*epmixf)(Aplcb, double, Aplcb); /* ravel */
     Aplcb (*epmixg)(Aplcb, int, Aplcb); /* Sel. spec. */
} Mixed_monadics;
typedef union mixed_dyadics {
     Aplcb (*epdix)(Aplcb, Aplcb);
     Aplcb (*epdixa)(Aplcb, Aplcb, int); /* with axis */
     Aplcb (*epdixe)(Aplcb, Aplcb, Aplcb); /* with axes */
     Aplcb (*epdixf)(Aplcb, Aplcb, Aplcb, Aplcb); /*sel.sp.*/
     Aplcb (*epdixg)(Aplcb, Aplcb, Aplcb, char *); /* pick */
} Mixed_dyadics;
```

FUNCTION ESTABLISHMENT ⎕FX **Lfx**

Quad fx is a monadic system function that accepts a representation in character matrix form of a defined function, and makes that function available for use. A nice enhancement in APL2, and also in CAPLIB2, is the ability to accept a nested vector containing simple character vectors, each describing a line of the defined function:

APL2 implements a left argument to Quad fx, consisting of a four-item boolean vector giving what it calls **execution properties.** This is not implemented in CAPLIB2.

APL Usage

```
apl> Lfx 2 7R'z#sum x z#+/x '      apl> Lcr 'sum'

sum                                z#sum x
                                    z#+/x

sum 1 3 5 7 9

25

apl> Lfx 'z#fmt r' 'z#Fr'          apl> fmt 'abcdef'

fmt                                abcdef

apl> Lfx 'z#fn r' 'z#1+rX2' Lav[1]

^ Unknown character.
Lfx 'z#fn r' 'z#1+rX2' Lav[1]
^ 58 - syntax error.
```

C Usage

See the example given on page 62

See Also

Assign (#), Canonical Representation (Lcr), Defined Function, Plus (+), Reduction (/), Reshape(R).

FUNCTION HEADER

The first thing in a defined function is always a **function header**, which begins with a template describing the function name and arguments. This is analogous to the *function declarator* that appears at the front of a program written in C. The template has one of six forms:

Type	With Result	Without Result
Dyadic	r # x f y	x f y
Monadic	r # f y	f y
Niladic	r # f	f

where:

- **f** is the name by which the defined function is known.
- **r** is the localized name of the variable that will contain the result. At least once during each execution of the defined function, an assignment must be made to this name.
- **x** is the localized name of the left argument of a dyadic defined function.
- **y** is the localized name of the rigth argument of a dyadic or monadic defined function.

See Also

Assign (#), Defined Function.

)HOST

IBM's APL2 has a **)HOST** system command that allows you to execute host system commands from within APL2. This is not yet implemented in CAPLIB2().

FUNCTION STOP △ H

For debugging purposes, you can stop any defined function and receive control just before any particular statement gets executed. To stop defined function **demo** before lines 5 and 9 are executed, for example, enter the APL statement -

```
sHdemo # 5 9
```

You'll receive control in the localized environment of **demo**. If it localized variable **a** and **b**, then you'll have access only to those local instances. You can interrogate and change any variable you have access to. You can resume execution at any statement **n** in **demo** by entering `$n`, or you can terminate **demo's** execution by entering tt $0.

See Also

Assign (#), Branch ($), Defined Function, Function Execution, Function Trace,)SI.

FUNCTION TRACE △ H

For debugging purposes, you can trace any statement in a defined function. You'll receive a report on your display each time a designated statement in a defined function is executed. To trace lines 1 thru 5 in defined function **demo**, for example, enter the APL statement -

```
tHdemo # I5
```

The function name, the number of the statement in brackets, and the final value of the statement is displayed each time it is executed. a branch statement is indicated by a right arrow ($ in CAPLIB2) followed by the number of the next statement to be executed.

See Also

Assign (#), Branch ($), Defined Function, Function Execution, Function Stop, Index Generator (I),)SI.

NOT LESS ≥ Z

For real numbers, returns 1 if the left argument is greater than or equal to the right argument minus **comparison tolerance Lct**. Returns 0 otherwise.

For complex numbers, the results of comparing their magnitudes is returned.

APL Usage

```
apl> 1 2 3 Z 3 2 1

0 1 1
```

C Usage

Subroutines **dblge, dblgep, ge, gep, ige**, and **igep** are related. They differ in the data types of arguments accepted, and the value returned. Subroutine **ge** for example accepts arguments of type **double**, and returns a value of type **int**.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
```

```
main()
{
     Dblge; Dblgep; Dyadic; Endoper;
     Ge; Gep; Idyadic; Ige; Igep;
     Litvect; Quadout; Relative;
     double left[2],rite[2],ret[2];
     int ileft,irite,iret;
     #include "aplinit.h"

     endoper(quadout(dyadic(dblge,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(relative(ge,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(idyadic(ige,
          litvect("1 2 3"),
          litvect("3 2 1"))));
     *left = 2.1;
     *rite = 5.6;
     dblgep(left,rite,ret);
     printf("dblgep returned %f\n",*ret);
     gep(left,rite,&iret);
     printf("gep returned %d\n",iret);
     ileft = 2;
     irite = 5;
     igep(&ileft,&irite,&iret);
     printf("igep returned %d\n",iret);
}

0 1 1
0 1 1
0 1 1
dblgep returned 0.000000
gep returned 0
igep returned 0
```

See Also

Relational Functions.

GETCB

C program usage only, creates a structure called an **aplcb** (see page 29) describing an APL variable. Storage for this **aplcb** and any storage it points to, is dynamically allocated in **getcb**, and can later be freed by calling subroutine **endoper** (see page 115). Parameters to **getcb** are as follows:

```
Aplcb getcb(dataptr,datacnt,datatyp,rank,dimptr)
void *dataptr; /* Pointer to data. */
int  datacnt,  /* Count of data items. */
     datatyp,  /* Data type. */
     rank,     /* Rank. */
     *dimptr;  /* Pointer to dimensions. */
```

Dataptr should point to an array of zero or more data items, the type of which depends on the value of **datatyp**. **Getcb** will copy this data into dyamically allocated storage, a pointer to which will be stored in the **aplcb** in field **aplptr**.

Datacnt should contain a count of the data items pointed to by **dataptr**.

Datatyp should be set to one of five values (see **#define**s on page 29) that determine the type of data pointed to by **dataptr**: **APLCHAR** `*char`; **APLINT** `*int`; **APLNUMB** `*double`; **APLCPLX** `*double`; and **APLAPL**. **APLAPL** is a specialized data type for performing indexing. See also **indexing, indexm**, and **setix**.

Rank is 0 for a scalar, 1 for a vector, 2 for a matrix, etc.

Dimptr may point to an array of **int** containing the dimensions for the new variable. **Getcb** will copy this array into dynamically allocated storage, pointed to by field **apldim** in the newly created **aplcb**. For scalars and vectors, this pointer may be **NULL**. In the case of scalars, there are no dimensions, and **datacnt** gives the single dimension for vectors.

Example of use

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Getcb; Indexm; Ivalue; Litvect; Perm;
     Quadout; Reshape; Setix; Shape; Temp;
     Aplcb m,n;
     #include "aplinit.h"
```

```
    m = perm(
        quadout(
            reshape(
                litvect("2 4"),
                litvect("3 1 4 2 1 4 4 1")
            )
        )
    ); /* ---------------------------------------> 3 1 4 2 */
    n = getcb(NULL, /* dataptr */               /* 1 4 4 1 */
        ivalue(shape(shape(m))), /* datacnt */
        APLAPL + APLTEMP, /* datatyp */         /* 4 3 2 1 */
        1,NULL); /* rank, dimptr */             /* 3 2 2 3 */
    setix(n,1,NULL); /* index to 1st dimension */
    setix(n,2,m); /* index to 2nd dimension */
    endoper(quadout(indexm(m,n,NULL))); /* ------> 4 1 1 4 */
    endoper(temp(m));                           /* 1 1 1 1 */
}
```

See Also

APLCB, ENDOPER.

GETCHRV

C program usage only, creates a structure called an **aplcb** (see page 29) describing an APL variable for character data **getchrv** will obtain from standard input. The sole argument is a character string to be sent to standard output as a prompt.

Example of use

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
    Aplmsg; Endoper; Execute; Getchrv;
    Aplcb out;
    extern int aplerr;
    char *cp;
    int i;
```

```
     #include "aplinit.h"

     for (;;) { /* loop until break */
          aplerr = 0; /* reset any prior error */
          out = getchrv("APL> "); /* get APL statement */
          if (aplerr) {
               fprintf(stderr,"%s\n",aplmsg(aplerr));
               exit(aplerr);
          }
          cp = out -> aplptr.aplchar;
            /* Point to input line. */
          for (i = out->aplcount; i; i--)
            putchar(*cp++); /* Echo. */
          printf("\n"); /* skip to a new line */
          if (out->aplcount == 0)
               break;
                 /* An empty line signals end of file. */
          if (out->aplcount == 1
            && *out->aplptr.aplchar == '$')
               break;
                 /* '$' also indicates end of file. */
          endoper(execute(out));
     }
     endoper(out); /* free $ APL variable */
}
```

Following was seen on my screen when I ran **sample39** on my PC. Text I entered on my keyboard shows immediately to the right of each **APL**> prompt:

```
C:> sample39

APL> 3X4

 12

APL> perimeter # 2 X (3 + 4)

APL> perimeter

 14
```

```
APL> x # 2 + L # 3 X y # 4

 12

APL> x

 14

APL> y

 4
```

See Also

APLCB, Assign (#), Plus (+), Quad Output (L), Times (X).

GETTEXT

C program usage only, obtains a line of text from the standard input. Returns the length of the line obtained, or -1 to indicate end-of-file. Recognizes the backspace, and will delete the most recently encountered character, and backs the input cursor up if standard input is coming from the keyboard.

Example of use

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
#define MAXLINE 80
main()
{
Gettext;
     int linelen;
     char line[MAXLINE]="";
     #include "aplinit.h"

     while ( -1 != (linelen = gettext(line, MAXLINE)))
         printf("line = '%s'\n",line);
}
```

See Also

GETVECT.

GETVECT

C program usage only, **getvect** sends a prompt to standard output, and obtains from standard input via a call to **gettext** (see page 165) a character string containing zero or more floating point constants. The string is edited by **getvect**, and the user prompted for corrections if necessary. When all constants are valid, they are converted to internal double floating point. An **aplcb** (see page 29) is initialized to point to this vector, and a pointer to the **aplcb** is returned.

Example of use

Program **poly** shown below uses matrix algebra to solve for coefficients of any polynomial of any degree **n**, given the **x** and **y** values for **n + 1** unique points on the line graph of that polynomial. **Poly** prompts you to enter the **x** and **y** values, then prints the solution, if it exists.
A typical dialogue with **poly** would be:

```
POLY:  Enter X values:

USER:  -1.5 -1 0 2.1

POLY:  Enter Y values:

USER:  -0.875 0 -2 0.961

POLY:  -2 -3 0 1
```

A similar dialogue with the APL interpreter would be:

```
APL>  x # -1.5 -1 0 2.1

APL>  y # -0.875 0 -2 0.961

APL>  y M x J . * -1 + I R x

-2 -3 0 1

APL>
```

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
```

```
main()
{
     Aplmsg; Getvect; Indxsub; Mdivide;
     Outrprd; Power; Quadout; Real;
     Aplcb c,w,x,y;
     #include "aplinit.h"

     indxorg = 0;
     x=getvect("Enter X values");
     y=getvect("Enter Y values");
     c=quadout(mdivide(y,outrprd(power,
         x,real(indxsub(*(x->apldim))))));
}
```

See Also

Assign (#), Exponential (*), Index Generator (I), Matrix Divide (M), Outer Product (.), Plus (+), Shape (R).

GRADE ⍋ ⍒ A W

Grade up (A v) grades **v** in ascending order, and **grade down (W v)** grades **v** in descending order. Classic APL allowed the use of the grade up and down functions only in monadic form. APL2 and CAPLIB2 both allow dyadic use as well.

Monadic Grade Up/Down

To me, the obvious result of the **grade** functions would be the items of argument **v** arranged in the proper collating sequence. This is not the case. Rather, the yield of **grade up** is a vector whose elements are the index of the smallest item of **v**, followed by the index of the second smallest item of **v**, etc. Thus what I think of as the obvious result might be produced by the APL expression **(v** [**A v**]**)**. If **v** contains **8 3 7 5**, then **(A v)** yields **2 4 3 1**, and **(v** [**A v**]**)** yields **3 5 7 8**.

If all items in **v** aren't unique, the ranking among equal elements is determined by their position. If **v** contains **4 3 1 3 4 2**, then **(A v)** yields **3 6 2 4 1 5**.

The result of monadic grade up or down is a permutation of **I1YRright** such that **C'right[Wright',((-1+RRright)R';'),']'** is a sorted ordering of the subarrays along **right's** first axis.

```
apl> A  a  # 7 2 5 11 3

2 5 3 1 4

apl> W  a

4 1 3 5 2

apl> " The index origin influences the generated indices
apl> Lio  #  0
apl> A a

1 4 2 0 3

apl> W a

3 0 2 4 1

apl> Lio#0
apl> Wa#23 11 13 31 12

3 0 2 4 1

apl> Lio#1
apl> Wa

4 1 3 5 2

apl> a[Wa]

31 23 13 12 11

apl> W23 14 23 12 14

1 3 2 5 4

apl> L#b#5 3 R 4 16 37 2 9 26 5 11 63 3 18 45 5 11 54

4 16 37
2  9 26
5 11 63
3 18 45
5 11 54

apl> Wb

3 5 1 4 2

apl> C'b[Wb',((-1+RRb)R';'),']'

5 11 63
5 11 54
4 16 37
3 18 45
2  9 26

apl> b[Wb;]

5 11 63
5 11 54
4 16 37
3 18 45
2  9 26

apl> c#4 23 54 28 2 11 51 26
apl> c#c,4 29 17 43 3 19 32 41
apl> c#3 2 4Rc,4 23 54 28 1 25 31 16
apl> c

4 23 54 28
2 11 51 26

apl> Wc

2 1 3
```

```
4 29 17 43
3 19 32 41

4 23 54 28
1 25 31 16
```

```
apl> c[Wc;;]

4 29 17 43
3 19 32 41

4 23 54 28
2 11 51 26

4 23 54 28
1 25 31 16
```

Dyadic Grade Up/Down

The argument **v** to monadic grade up or grade down must be a numeric vector. In classic APL, if you wanted to sort character vectors, you did so indirectly using an APL System Variable called the **atomic vector** . This is a 256-element character vector named **Lav** containing all possible characters in the collating sequence of the local environment. For example, if **a** is a vector containing **"cbaex"**, then the APL expression **(a** [**A Lav I a**]**)** yields **"abcex"** (**I** in this expression refers to the **Index Of** function described on page 182).

APL2 and CAPLIB2 makes this a littler simpler by allowing you to specify as the left argument to the grade functions, a character array indicating the collating sequence to use.

Items of the right argument not found in the left have collating sequences as if they were found at the end of the left argument.

The last axis of the left argument is the most significant for collating, and the first is the least.

The relative order of identical subarrays is unchanged.

```
apl> Lio#0
apl> 'abcde' W 'bead'

1 3 0 2
```

```
apl> Lio#1
apl> 'abcde' W 'bead'

2 4 1 3
```

```
apl> L#a#5 4R'deadbadecedebeaddeed'

dead
bade
cede
bead
```

```
apl> 'abcde'Wa

5 1 3 4 2
```

```
deed

apl> bWc

1 4 3 2 5
```

```
apl> c#'face$'
apl> b#'@$&abcdef'
apl> c[bWc]

feca$
```

```
apl> k#5 4R'dealDealdeadDeadDEED'
apl> h#2 12R'abcdefghijklABCDEFGHIJKL'
apl> k

deal
Deal
dead
Dead
DEED

apl> k[z;]

DEED
Deal
deal
Dead
dead
```

```
apl> h

abcdefghijkl
ABCDEFGHIJKL

apl> L#z#hWk

5 2 1 4 3
```

```
apl> locase#'abcdefghijklmnopqrstuvwxyz'
apl> hicase#'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
apl> dcs#' ',hicase,[.5]locase
apl> dcs#(dcs,[1]((8,-1YRdcs))R' '),,F10 1R-1+I10
apl> dcs

 ABCDEFGHIJKLMNOPQRSTUVWXYZ0
 abcdefghijklmnopqrstuvwxyz1
                           2
                           3
                           4
                           5
                           6
                           7
                           8
                           9
```

```
apl> dcsW'avenue'

2 5 4 3 6 1

apl> h[L#dcsWh#'yzommxa']

2 1 6 3 4 5 7

zyxomma
```

```
apl> q[L#dcsWq#5 4R'sent zapdownalsoboa ';]
```

```
1 3 5 4 2

sent
down
boa
also
 zap
```

```
apl> 'abcde'W'dabbed'

5 1 6 3 4 2

apl> q[L#'abcde'Wq#'bleat']

2 5 3 1 4

lteba
```

```
apl> k[L#dcsWk#5 4R'dealDealdeadDeadDEED';]

5 1 2 3 4

DEED
deal
Deal
dead
Dead

apl> L#s # ‘X 'X1' 'X10' 'X2' 'X21' 'X3' 'X9' 'X11' 'x3'

X1
X10
X2
X21
X3
X9
X11
x3

apl> Lio#0
apl> 'abcde'A'bead'

2 0 3 1

apl> L#a#5 4R'deadbadecedebeaddeed'

dead
bade
cede
```

```
apl> s[L#dcsWs;]

4 7 2 6 8 5 3 1

X21
X11
X10
X9
x3
X3
X2
X1

apl> Lio#1
apl> 'abcde'A'bead'
```

```
bead
deed

apl> c#'face$'
apl> b#'@$&abcdef'
apl> bAc

5 2 3 4 1

apl> dcs A 'AVENUE'

1 3 6 4 5 2
3 1 4 2

apl> 'abcde'Aa

2 4 3 1 5

apl> c[bAc]

$acef

apl> dcs A h#'LWLOIBY'

6 5 1 3 4 2 7

apl> h[dcs A h]

BILLOWY

apl> L # m # dcs A L # k # 5 4 R 'sent zapdownalsoboa '

sent
 zap
down
also
boa

2 4 5 3 1
apl> k[m;]

 zap
also
boa
down
sent

apl> k#5 4R'dealDealdeadDeadDEED'
apl> h#2 12R'abcdefghijklABCDEFGHIJKL'
apl> k

deal
Deal
dead
Dead
DEED
apl> h

abcdefghijkl
ABCDEFGHIJKL

apl> L#z#hAk

3 4 1 2 5

apl> k[z;]
```

```
dead
Dead
deal
Deal
DEED

apl> L#s # ‘X 'X1' 'X10' 'X2' 'X21' 'X3' 'X9' 'X11' 'x3'

X1
X10
X2
X21
X3
X9
X11
x3

apl> 'abcde'A'dabbed'

2 3 4 1 6 5

apl> L#w#'abcde'A'exact'

3 4 1 2 5
```

```
apl> s[L#dcsAs;]

1 3 5 8 6 2 7 4

X1
X2
X3
x3
X9
X10
X11
X21

apl> 'exact'[w]

acext
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Gradedn; Gradedn2;
     Gradeup; Gradeup2; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(
       gradedn(litvect("4 3 1 3 4 2"))));
     endoper(quadout(
       gradeup(litvect("4 3 1 3 4 2"))));
     endoper(quadout(
       gradedn2(chrvect("abcde"),chrvect("bead"))));
```

```
     endoper(quadout(
       gradeup2(chrvect("abcde"),chrvect("bead"))));
}
```

```
1 5 2 4 6 3
3 6 2 4 1 5
2 4 1 3
3 1 4 2
```

See Also

Assign (#), Index Of(I), Index Origin (Lio).

)GROUP

)group grpname obj1 obj2 ···

The named objects are grouped and can later be collectively referenced by their group name in)COPY,)PCOPY,)DROP and)ERASE commands. This makes it convenient to copy into the active workspace a group of related items, and later discard them by their group name.

If **grpname** is already in use as a group name, the new list replaces the old. If **grpname** is otherwise already in use, you'll receive an error message.

The list may be empty, in which case the group is said to be **dispersed**, and a subsequent reference to the group name in e.g. an)ERASE command has no effect.

If a name in the list is the group name itself, the former membership is enhanced with the names in the present list.

See Also

Appendix E.

)GRP

)grp grpname

Reports the membership of the named group.

See Also

Appendix E.

)GRPS

)grps [**a**]

Reports in alphabetical order, starting with the indicated letter if given, a list of the groups in the active workspace.

See Also

Appendix E, Name List (Lnl).

GREATER > >

For real numbers, returns 1 if the left argument is greater than the right argument plus **comparison tolerance Lct**. Returns 0 otherwise.

For complex numbers, the results of comparing their magnitudes is returned.

APL Usage

```
apl> 1 2 3 > 3 2 1

0 0 1
```

C Usage

Subroutines **dblgt, dblgtp, gt, gtp, igt**, and **igtp** are related. They differ in the data types of arguments accepted, and the value returned. They all, however, implement the **greater than** comparison operation.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
Dblgt; Dblgtp; Dyadic; Endoper;
     Gt; Gtp; Idyadic; Igt; Igtp;
     Litvect; Quadout; Relative;
     double left[2],rite[2],ret[2];
     int ileft,irite,iret;
     #include "aplinit.h"

     endoper(quadout(dyadic(dblgt,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(relative(gt,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(idyadic(igt,
          litvect("1 2 3"),
          litvect("3 2 1"))));
     *left = 2.1;
     *rite = 5.6;
     dblgtp(left,rite,ret);
     printf("dblgtp returned %f\n",*ret);
     gtp(left,rite,&iret);
     printf("gtp returned %d\n",iret);
     ileft = 2;
     irite = 5;
     igtp(&ileft,&irite,&iret);
     printf("igtp returned %d\n",iret);
}
```

```
0 0 1
0 0 1
0 0 1
dblgtp returned 0.000000
gtp returned 0
igtp returned 0
```

See Also

Relative Functions.

IDENTITY

The APL functions **inner product, reduce,** and **scan** all involve applying scalar dyadic functions such as **plus (+)** and **times (X)** to adjacent pairs of items from their arguments. But what result does one get when applying these functions to empty vectors? The answer is an **identity**. A **left identity element z** for dyadic function **F** is such that **z F x** equals **x**, and a **right identity element z** is such that **x F z** equals **x**.

Function		Identity	Left/Right
And	&	1	Left, Right
Binomial	!	1	Left
Circle	O		None
Divide	%	1	Right
Equal	=	1	Left, Right
Greater	>	0	Right
Less	<	0	Left
Logarithm	@		None
Maximum	S	(note 1)	Left, Right
Minimum	D	(note 2)	Left, Right
Minus	_	0	Right
Nand	P		None
Nor	‘V		None
Not equal	∧ =	0	Left, Right
Not greater	K	1	Left
Not less	Z	1	Right
Or	V	0	Left, Right
Plus	+	0	Left, Right
Power	*	1	Right
Residue	|	0	Left
Times	X	1	Left, Right

- Note 1: The largest in magnitude of representable negatives.
- Note 2: The largest representable number.

Table 2.2: Identity Elements of Primitive Scalar Dyadic Functions

IDYADIC

Extends dyadic integer scalar functions such as **iplus** and **iminus** to conformable pairs of APL variables containing integer data item-by-item. If the APL variables aren't for integer data, **idyadic** will do the necessary conversions.

The arguments must either agree in shape, or one must have only a single element, which is repeatedly matched with elements of the other. If both arguments have one item, but differing ranks, the result has the higher rank.

See Also

Numerous examples of the use of **idyadic** exist throughout this manual. See, for example, IPLUS, IMINUS, IDIVIDE. See also the closely related function DYADIC.

)IN

IBM's APL2 has a **)IN** system command that allows you to read what are called "Transfer Files" into the active workspace. This is not yet implemented in CAPLIB2(), but the **)read** CAPLIB2 system command is similar.

IMONADIC

Extends monadic integer scalar functions such as **factorial** and **absolute value** to an APL variable containing integer data item-by-item. If the APL variable isn't for integer data, **imonadic** will do the necessary conversions.

See Also

Numerous examples of the use of **imonadic** exist throughout this manual. See, for example, IFACTORL, INOT, and IABS. See also the closely related function MONADIC.

INCLUDES.H

Most of the sample C programs in this book start with two lines similar to these:

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
```

For the most part **INCLUDES** offers a shorthand way of specifying the includes necessary for your program. In traditional C program notation, the above might be specified as -

```
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
#include "aplcb.h"
#include "aplenv.h"
#include "aplext.h"
```

This leaves a lot of unnecessary white space, and doesn't contribute to the documentation of what the program does. I try to keep source files under 60 lines so they'll stay on one page on most printers. Using the traditional notation, sometimes a third of those lines can be #include statements, which doesn't leave much room for program logic.

Also, #include statements often need to be done in a certain order. Stdlib.h, for example, usually needs to be first. With INCLUDES, I can indicate what's necessary in a sorted list, so I can easily eyeball it to check for a particular name, and then let the logic in "includes.h" ensure the order they're actually included is correct.

Also, certain includes must be done every time. Stdlib.h, aplenv.h (see page 39), and entrys.h (see page 119) are examples. It doesn't contribute to the clarity of programs to have all these listed at the top of every program you write.

There are twenty six items you can specify on the **#define INCLUDES** statement, but only twenty two are important to C programmers using the CAPLIB2 subroutine library:

- APLCB: Generates a "**#include "aplcb.h"**", described on page 29).
- APLCHDEF: Generates a "**#include "aplchdef.h"**", described on page 31.
- APLDEBUG: Generates a "**#include "apldebug.h"**", described on page 33.
- APLDERIV: Generates a "**#include "aplderiv.h"**", described on page 33.
- APLDOS: If the environment is DOS, indicated by a switch in **aplenv.h** (see page 39), this will generate a "**#include <dos.h>**."

- APLFUNCI: Generates a “**#include “aplfunc.h”**”, described on page 41.
- APLMAIN: Indicates the main C program. Only one C program in CAPLIB^2, namely APL.C, specifies APLMAIN. It is the main program for the CAPLIB^2 APL interpreter. Specifying APLMAIN triggers things like the declaration of all external variables that are global to all CAPLIB^2 subroutines (See “aplext.h” on page 39).
- APLMEM: Generates a “**#include “aplmem.h”**”, described on page 44. Redefines malloc and free respectively to aplmall and aplfree. Used in debugging.
- APLTOKCD: Generates a “**#include “apltokcd.h”**”, described on page 45.
- APLTOKEN: Generates a “**#include “apltoken.h”**”, described on page 46.
- CONIO: Generates a “**#include <conio.h>**”, if the current environment is DOS, indicated by switch APL_DOS in aplenv.h.
- ERRNO: Generates a “**#include <errno.h>**”.
- FUNCODES: Generates a “**#include “funcodes.h”**”, described on page 151.
- FUNSTRUC: Generates a “**#include “funstruc.h”**”, described on page 155.
- IO: For DOS, generates a “**#include <io.h>**”, and otherwise generates a “**#include <asm/io.h>**.”
- LIMITS: Generates a “**#include <limits.h>**.’
- MATH: Generates a “**#include <math.h>**.’
- STAT: For DOS, generates a “**#include <sys\stat.h>**”, and otherwise generates a “**#include <sys/stat.h>**.”
- STDIO: Generates a “**#include <stdio.h>**.’
- STRING: Generates a “**#include <string.h>**.’ and for DOS, sets the preprocessor variable APL_STRCMP to “strcmpl” and otherwise to “strcmp.”
- TREE: Generates a “**#include “tree.h”**”, described on page 334.

- TRIGKEYS: Generates a "**#include "trigkeys.h"**", described on page 338.

See Also

Numerous examples of the use of using **includes.h** exist throughout this manual. See, for example, Appendix F, beginning on page 373. See also **entrys.h**, included automatically by **includes.h**.

INDEX GENERATOR ≀ I

Called **interval** in IBM's manual on APL2, for a positive integer argument **n**, CAPLIB2 expression **In** produces a vector of length **n** containing the first **n** integers in order, beginning with the value of the **index origin Lio** described on page 184.

APL Usage

```
apl> I5                          apl> I5+Lio#0

1 2 3 4 5                        0 1 2 3 4

apl> " I0 produces a vector of length 0
apl> RI0

0
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Indexgen; Iscalar; Quadout;
     #include "aplinit.h"

     indxorg = 1; /* index origin */
     endoper(quadout(indexgen(iscalar(5))));
     indxorg = 0; /* index origin */
     endoper(quadout(indexgen(iscalar(5))));
}
```

```
1 2 3 4 5

0 1 2 3 4
```

See Also

Assign (#), Index Origin (Lio), Plus (+).

INDEX OF ≀ I

Index of, v I s, yields for each item in **s** the index of its earliest occurrence in **v**. If an item of **s** can't be matched by an item in **v**, the result is **Lio + R v** (the index value is influenced by the current setting of the **index origin**).

For **v I s**, v may be any shape you care to make it. **Index of** will still treat it as though it were a vector. The right argument, **s**, may also be any shape. The result of **v I s** will have the shape of **s**.

This CAPLIB2 expression produces 1 for any arrays **left** and **right**:

```
(left I right) ‘= Lio++/&\˜ right J.‘= left
```

APL Usage

```
apl> Lio#1
apl> a#'abcdefghijklmnopqrstuvwxyz '
apl> aIb#'head chief'
                                 apl> aI'var3'
8 5 1 4 27 3 8 9 5 6
                                 22 1 18 28

apl> aI2 5Rb                     apl> (3 9Ra)I2 5Rb

8 5 1 4 27                       8 5 1 4 27
3 8 9 5  6                       3 8 9 5  6

apl> left # 8 4 2 7 3
apl> rite # 3 8 4
apl> Lio#0                       apl> Lio#1
apl> left I rite                 apl> left I rite

4 0 1                            5 1 2
```

```
apl> Lio#0
apl> 'sport' I 'top'

4 2 1
```

```
apl> Lio#1
apl> 'sport' I 'top'

5 3 2
```

```
apl> Lio#1
apl> ((2 3) (I0) 'me') I 'me' (I0)

3 2

apl> 'wizard' I 'oz'

7 3

apl> 6 7 4 I 4 7 (I0)

3 2 4

apl> ('oh' 'no' 'i') I 'no' 'on'

2 4

apl> 5 5 8 8 9 I 8 9 5

3 5 1
```

```
apl> 8 9 5 I 2 5 8

4 3 1

apl> 'ohnoi' I 'no' 'on'

6 6
```

```
apl> 'banana' I 'banana'

1 2 3 2 3 2
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Indexof; Quadout;
     #include "aplinit.h"

     endoper(quadout(indexof(
          chrvect("abcdefghijklmnopqrstuvwxyz "),
          chrvect("head chief")
     )));
}
```

```
8 5 1 4 27 3 8 9 5 6
```

See Also

Assign (#), Index Origin (Lio), Indexing ([]), Reshape (R).

INDEX ORIGIN ☐IO **Lio**

The **index origin** is a system variable in APL used to control how elements within an array are indexed. Eleven APL functions are affected by the index origin value: **Deal; Grade Down; Grade Up; Index generator; Index of; Pick; Reduce; Roll; Scan; Squad;** and **Transpose**.

APL Usage

In APL the index origin is referred to by the system variable name **Lio**. You can assign a new value to **Lio**, but it should only be set to either 0 or 1, and is by default 1.

If vector **v** contains 4 5 6, v[1] produces 4 if **index origin** is 1, and produces 5 if it is 0.

```
apl> Lio#1
apl> r#34 18 24
apl> r[1]

34

apl> Lio#0
apl> r[1]

18

apl> r[0]

34

apl> Lio#1
apl> Ar

2 3 1

apl> Lio#0
```

```
apl> Ar

1 2 0
```

C Usage

In a C program, the index origin is referred to by the name **indxorg**, which must be declared **int** and **extern**.

See Also

Deal (?), Grade (A, W), Index Generator (I), Index Of (I), Indexing ([]), Reduce (R), Scan (\ ), and Transpose (T).

INDEXING

Indexing refers to the process of selecting items from a vector or array. There are two ways, syntactically different, but functionally similar:

1. **Bracket Index**. For any vector **a, a[i]** selects items from **a**. To select items from a matrix the syntax would be **a[i;j],** and for higher arrays the number of semicolon-delimited arrays inside the brackets increases according to the rank of **a.** Any index array may be omitted, leaving two consecutive semicolons, implying all possible index values for that axis. Index values can be repeated, in which case the indicated item is also repeated.

2. **Squad**. APL2 and CAPLIB2 only, does not exist in classic APL. The CAPLIB2 expression "**i j k** '*Q* **a**" is equivalent to "**a[i;j;k]**." The rules for squad indexing are what you'd expect, given its functional equivalence to bracket indexing. The left argument must be a nested scalar or vector of arrays, where each array must consist only of positive integers in the range acceptable as index values. These rules also apply for **left 'Q right**:

   - The shape of the result is given by the CAPLIB2 expression -

     ```
     'X,/R'1 left
     ```

   - The length of **left** raveled (**R,left**) must equal the rank of **right** (**RRright**).
   - The rank of the result is given by the CAPLIB2 expression -

     ```
     ,+/ER'1R'1 left
     ```

The values used as indices are affected by the **index origin (Lio)** described on page 184, and are said to be **0-origin** or **1-origin**. If vector **x** contains 2 3 5 7 11, **x**[**4 1**] produces 7 2 if **Lio** is 1, but produces 11 3 if **Lio** is 0.

Indexing (squad) with axis

A bracketed set of axis indicators can be used with squad, similar in function to **bracket index** described above, with some indices omitted. This CAPLIB2 expression always produces 1 for any 3-dimensional array **a**:

```
a[i;;j] ‘= i j ‘Q[Lio+0 2] a
```

APL Usage

```
apl> " the shape of m[i;j] equals (Ri),Rj
apl> L # m # 3 4 R I 12   apl> m[2;3]    apl> m[1 3;2 3 4]

1  2  3  4                7               2  3  4
5  6  7  8                               10 11 12
9 10 11 12

apl> " indices may be omitted, see these matching results
apl> m[;2 3]               apl> m[I1YRm;]

 2  3                       2  3
 6  7                       6  7
10 11                      10 11

apl>" indexed variables can be assigned values
apl> m[3;2 3] # 0          apl> m[3;2 3] # 98 99
apl> m                     apl> m

1 2 3  4                   1  2  3  4
5 6 7  8                   5  6  7  8
9 0 0 12                   9 98 99 12

apl> 2‘Q[1]2 3 4R’abcdefghijklmnopqrstuvwxyz’

mnop                       apl> v#2 2.3 -5 999 .01
qrst                       apl> 3‘Qv
uvwx
                           -5
```

```
apl> (I0)‘Q1 " squad can index a scalar.

1

apl> (‘Z2 3R1 2 1 4 1 2)‘Qv

  2 2.3 2.0
999 2.0 2.3

apl> 3(1 3)‘Qm

9 11

apl> (2 3)(,4)‘Qm

 8
12

apl> R(I0)(I0)‘Qm

0 0

apl> Lio#0
apl> ’curtail’[0 1 3]

cut

apl> L#z#a[2 5 1]

9 -3 23

apl> Rz

3

apl> L#b#2 3R1 4 3 2 6 5
```

```
apl> (‘Z3 4)‘Qv

-5 999

apl> m#3 4RI12
apl> 3 1‘Qm

9

apl> (2 3)4‘Qm

8 12

apl> R(1 2)(3 4R3)‘Qm

2 3 4

apl> a#23 9 6.3 8 -3 7
apl> L#z#a[3]

6.3

apl> Lio#1
apl> ’curtail’[1 2 4]

cut

apl> L#c#3 6R’abcdefghijklmnopqr’

abcdef
ghijkl
mnopqr

apl> L#j#c[2;3]

i
```

```
1 4 3
2 6 5

apl> L#q#a[b]

23 8  6.3
 9 7 -3.0

apl> Rq

2 3

apl> ’nab’[3 2 1 2 1 2]

banana

apl> L#h#2 4R3 4 1 2 2 3 4 1

3 4 1 2
2 3 4 1

apl> ’emit’[h]

item
mite

apl> L#v#’h’ ’hi’ (’him’ ’his’)

 h hi   him hiz

apl> L#z#v[1]

 h

apl> ‘=z
```

```
apl> Rj

1

apl> L#p#c[1;3 1 4]

cad

apl> Rp

3

apl> L#m#c[1 2;1 3]

ac
gi

apl> Rm

2 2

apl> L#n#c[1 3;2 3R6 1 3 4 5 2]

fac
deb

rmo
pqn

apl> L#d#3 4Rc[1 2;]

abcd
efgh
ijkl

apl> d[;1]

aei
```

```
1

apl> RRz

0

apl> L#e#v[2]

 hi

apl> ‘=e

2

apl> L#s#v[3]

  him his

apl> ‘=s

3

apl> RRs

0

apl> d[3;]

ijkl

apl> L#n#c[1 3;2 3R6 1 3 4 5 2]

fac
deb

rmo
pqn

apl> Rn

2 2 3

apl> L#u#2 3 4R(,c),’stuvwx’

abcd
efgh
ijkl

mnop
qrst
uvwx
```

```
apl> " Rx[i] equals Ri
apl> a # ’abcdefg’
apl> L # i # 4 3 R 3 1 4 2 1 4 4 1 2 4 1 4

3 1 4
2 1 4
4 1 2
4 1 4

apl> a[i]

cad
bad
dab
dad

apl> u[1;2;4]

h

apl> u[2;1;1 3 4]

mop

apl> u[;2;4]

ht
```

```
apl> u[1;;3]              apl> u[1;1 3;2 4]

cgk                       bd
                          jl
apl> u[2;1;]
                          apl> u[;3;]
mnop
                          ijkl
                          uvwx
apl>a#2 3 4RI24
apl> 2‘Q[1]a              apl> a[2;;]

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

apl> (1 3)4‘Q[2 3]a       apl> a[;1 3;4]

 4 12                      4 12
16 24                     16 24
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Getcb; Indexm; Ivalue; Litvect; Perm; Quadout;
     Reshape; Setix; Shape; Temp;
     Aplcb m,n;
     #include "aplinit.h"

     m = perm(
          quadout( /* --------------------> 3 1 4 2 */
               reshape( /*                  1 4 4 1 */
                    litvect("2 4"),
                    litvect("3 1 4 2 1 4 4 1")
               )
          )
     );
     n = getcb(NULL, /* dataptr */
          ivalue(shape(shape(m))), /* datacnt */
```

```
          APLAPL + APLTEMP, /* datatyp */
          1,NULL); /* rank, dimptr */
     setix(n,1,NULL); /* index to 1st dimension */
     setix(n,2,m); /* index to 2nd dimension */
     endoper(quadout(indexm(m,n,NULL))); /* ---> 4 3 2 1 */
     endoper(temp(m)); /*                        3 2 2 3 */
/* */
/*                                               4 1 1 4 */
/*                                               1 1 1 1 */
}
```

See Also

Assign (#), Index Generator (I), Index Origin (Lio), INDEXM, Quad Output (L), Reshape (R), and SETIX.

INDEXM

CAPLIB[2] subroutine **indexm** enables a C language program to perform indexing of APL vectors and arrays. **Indexm** declares its arguments as follows:

```
struct aplcb *indexm(left,indices,rite)
struct aplcb *left,*indices,*rite;
```

Left points to an APL variable from, and perhaps to, which values are to be indexed.

Indices identifies the index values to be used, and determines the shape of the output. These values are considered to be either **0-origin** or **1-origin** according to the setting of external variable **indxorg**, which contains the **index origin** currently in force.

If **rite** isn't **NIL**, it contains values to be stored in the indexed elements of **left**.

Indexm differs from **indexv** in that left may be of any shape, and the output may have rank other than one. **Indexm** is the true implementation of the APL indexing function.

See the programming example under the heading **INDEXING** on page 185.

See Also

Index Origin (Lio), Indexing ([]), INDEXV, and SETIX.

INDEXV

Indexv is a simplified version of CAPLIB[2] subroutine **indexm**. It declares its arguments as follows:

```
struct aplcb *indexv(left,indices,rite)
struct aplcb *left,*indices,*rite;
```

Left points to an APL variable from, and perhaps to, which values are to be indexed.

Indices identifies the index values to be used. The actual shape of **left** is ignored, and the indices must be chosen as though left were a vector, which is the shape of the output of **indexv**.

If **rite** isn't **NIL**, it contains values to be stored in the indexed elements of **left**.

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Indexv; Iscalar; Litvect; Quadout;
     #include "aplinit.h"

     indxorg = 1; /* index origin */
     endoper(
          quadout( /* -------------------------> 10 */
               indexv(
                    litvect("10 20 30"),
                    iscalar(1),
                    NULL)));
     indxorg = 0; /* index origin */
     endoper(
          quadout( /* -------------------------> 20 */
               indexv(
                    litvect("10 20 30"),
                    iscalar(1),
                    NULL)));
}
```

See Also

Index Origin (Lio), Indexing ([]), INDEXV, and SETIX.

INDXSUB

C program usage only, **indxsub** is a fast interface to the **index generator** function. It declares itself as follows:

```
Aplcb indxsub(indx)
int indx;
```

Indxsub(3) for example, returns a pointer to an aplcb for the vector 1 2 3 if indxorg=1, and the vector 0 1 2 if indxorg=0. **Indxsub(-3)** returns the vector 3 2 1 if indxorg=1, and 2 1 0 if indxorg=0.

Example of use

Numerous examples exist throughout this manual of using **indxsub**. See for example the descript of the **getvect** on page 166.

See Also

Index Generator (I).

INNER PRODUCT . .

The general form of **inner product** is **aF.Gb**, **a** and **b** being variables, **F** and **G** being primitive scalar dyadic functions such as **plus (+)** and **minus (_)**. The APL expression **a+.Xb** is an example of **inner product**. For vectors, this is equivalent to **+/aXb**, and is also known by mathematicians as the *dot product*, or *scalar product*. **Inner product** is extended to arrays along the last axis of the first argument, and first axis of the last. Given two matrices, **a+.Xb** is equivalent to the familiar **matrix product** of linear algebra.

Either argument of **inner product** may be a scalar or one-element vector, in which case it is extended as required by the shape of the other argument.

APL Usage

```
apl> L#a#3 3R2 -1 1 3 1 2 2 3 -1
                                apl> L#b#3 1 R 3 4 5
2 -1  1
3  1  2                         3
2  3 -1                         4
                                5
apl> a +.X b
                                apl> 3 -1 -2 +.X 2 -3 .5
     7
    23                          8
    13
                                apl> k +.E '0123456789'
apl> (2 2R10XI4) +.X 2 3RI6
                                0 5 1
 90 120 150
190 260 330

apl> m &.= L # m # 4 4 R 1 1 1 1 0 1 1 1 0 0 1 1 0 0 0 1

1 1 1 1                         apl> L#p#2 2R(.1XL#j#3 2RI6)
0 1 1 1
0 0 1 1                         1 2
0 0 0 1                         3 4
                                5 6
0 0 0 1
0 0 0 0                          .1 0.2
0 0 0 0                          .3 0.4
0 0 0 0

apl> L # k # 3 8 R 'saturday7/04/99 july 4  '

saturday                        apl> j,.+p
7/04/99
july 4                            1.1 2.3   1.2 2.4
                                  3.1 4.3   3.2 4.4
apl> (2 3 R I 6) D.S 3 4 RI12     5.1 6.3   5.2 6.4

1 2 3 4                         apl> j+.,p
4 4 4 4
                                 3.4  3.6
```

```
apl> L#s#3 5R'sandybettygrace'

sandy
betty
grace

apl> 'sandy'&.=Ts

1 0 0

apl> L#q#(2 0R0)+.X0 4R5

0 0 0 0
0 0 0 0
```

```
 7.4  7.6
11.4 11.6

apl> s&.='sandy'

1 0 0

apl> Ru#(0 2R0)+.X2 0R0

0 0

apl> Rq

2 4
```

C Usage

$\mathrm{C_{APL}IB^2}$ subroutine **innrprd** declares itself in this fashion:

```
struct aplcb *innrprd(opera,operb,identity,left,rite)
void (*opera)(),(*operb)(),*identity;
struct aplcb *left,*rite;
```

Opera and operb are of course the two scalar functions, such as **plus** and **times**. **Left** and **rite** are pointers to the two APL variables. **Identity** elements are explained on page 177.

```
/* matrix multiply with integers */
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Indxsub; Innrprd; Iplus; Itimes;
     Litvect; Quadout; Reshape;
     int identity=0;
     #include "aplinit.h"

     endoper(
          quadout( /* -------------------------> 22 28 */
               innrprd(                      /* 49 64 */
                    iplus,
```

```
                    itimes,
                    &identity,
                    reshape(
                         litvect("2 3"),
                         indxsub(6)
                    ),
                    reshape(
                         litvect("3 2"),
                         indxsub(6)
                    )
               )
          )
     );
}

/* matrix multiply with real numbers */
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Indxsub; Innrprd; Litvect; Plus;
     Quadout; Reshape; Times;
     double identity=0.0;
     #include "aplinit.h"

     endoper(
          quadout( /* ----------------------> 26.62 33.88 */
               innrprd(                  /* 59.29 77.44 */
                    plus,
                    times,
                    &identity,
                    reshape(
                         litvect("2 3"),
                         litvect("1.1 2.2 3.3 4.4 5.5 6.6")
                    ),
                    reshape(
                         litvect("3 2"),
                         litvect("1.1 2.2 3.3 4.4 5.5 6.6")
                    )
               )
          )
     );
```

```
}

/* matrix multiply with complex numbers */
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Indxsub; Innrprdp; Litvect; Plusx;
     Quadout; Reshape; Timesx;
     double identity[]={0.0,0.0};
     #include "aplinit.h"

     endoper(
          quadout( /* ---------------------> 0j17 -6j20 */
               innrprdp(                 /* 6j20 0j26  */
                    plusx,
                    timesx,
                    identity,
                    reshape(
                         litvect("2 3"),
                         litvect("1j1 1j2 1j3 2j1 2j2 2j3")
                    ),
                    reshape(
                         litvect("3 2"),
                         litvect("1j1 1j2 2j1 2j2 3j1 3j2")
                    )
               )
          )
     );
}
```

See Also

Assign (**#**), Identities, Plus (+), Quad Output (L), Reshape (R), Times (X).

INTEGER

C programming usage only, **integer** returns a copy of the APL variable received as argument, converting to integer if the input is floating point, and indicating an error if it is character. In finding the integer value, the real number is truncated downwards.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Integer; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(integer(quadout(
       litvect("1.25 7.89 -3.45")))));
}
```

```
1.25 7.89 -3.45

1 7 -4
```

INVERT

C programming usage only, **invert** returns the inverse of its matrix argument. It declares itself as follows:

```
     double *invert(m,rows)
     double *m;
     int rows;
```

Argument **m** points to the double floating point array. Argument **rows** describes the number of rows and columns in the array pointed to by **m**. Subroutine **invert** calls **malloc** to allocate storage to contain the inverse of **m**, and returns a pointer to this storage, unless the matrix isn't invertible, in which case it returns **NULL**.

```
#define INCLUDES APLMAIN+STDIO
#include "includes.h"
main()
{
     Invert; Matmult;
     static double a[3][3] = {
          {11.0, 12.0, 13.0},
          {50.0,  0.0, 42.0},
          {37.0, 73.0, 26.0}};
     static double x[3] = {218.0, 544.0, 805.0};
     double *answer;
```

```
     answer=matmult(invert((double *)a,3),x,3,3,1);
     printf("answer=%4.2f %4.2f %4.2f",
       *answer,*(answer+1),*(answer+2));
     free(answer);
}
```

```
answer=5.00 6.00 7.00
```

IREDUCE ⌿ / ‘/ /

Ireduce is identical in function to **reduce**, described on page 275, except it deals only in integers.

APL Usage

```
+ / 1 2 3

    6
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Iplus; Ireduce; Litvect; Quadout;
     int axis=1,identity=0;
     #include "aplinit.h"

     endoper(quadout(ireduce(iplus,&identity,
       litvect("4 5 6"),axis)));
}
```

```
15
```

See Also

Reduce (/).

ISCALAR

C program usage only, creates an apl variable from an integer constant.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Iscalar; Quadout;
     #include "aplinit.h"

     endoper(quadout(iscalar(2)));
}

2
```

See Also

Numerous examples of using **iscalar** exist throughout this manual. See for example, Deal (?), Decode (B), Drop (U), Encode (N), IDIVIDE, Index Generator (I), Index Of (I), Take (Y).

ISCAN + \ ‘\ \

Iscan is identical in function to **scan**, described on page 307, except it deals only in integers.

APL Usage

```
apl> + \ 1 2 3 4 5

1 3 6 10 15
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Iplus; Iscan; Litvect; Quadout;
     int axis=1,identity=0;
```

```
     #include "aplinit.h"

     endoper(quadout(iscan(iplus,&identity,
       litvect("4 5 6"),axis)));
}

4 9 15
```

See Also

REDUCE, SCAN.

IVALUE

C program usage only, returns the integer value of the first or only element pointed to by the **aplcb** passed as argument. In addition to the example below, see also the example under the heading **indexing** on page 185.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Ivalue; Litvect;
     #include "aplinit.h"

     printf("Sample usage of ivalue = %d\n",
       ivalue(litvect("2 3 4")));
     printf("Sample usage of ivalue = %d\n",
       ivalue(litvect("3.5 3 4")));
}

Sample usage of ivalue = 2
Sample usage of ivalue = 3
```

See Also

VALUE.

LABELS : :

A **label** can appear on the front of any statement in a defined function except the function header. The label consists of a name separated from the statement body by a colon. The name becomes localized, and must be unique within the function. The name can be referenced, but it can't be assigned to. Its value is the statement number on which it resides.

APL Usage

```
apl> Gcompare[L]G

          G x compare y
    [1]     $biggerXIx>y
    [2]     $smallerXIx<y
    [3]     'x and y are equal'
    [4]     $0
    [5]    bigger:'x is bigger than y'
    [6]     $0
    [7]    smaller:'x is smaller than y'
          G

apl> 3 compare 2

    x is bigger than y

apl> 3 compare 3

    x and y are equal
```

See Also

Defined Function, Branching.

LAMINATE , ,

Joins two APL variables of identical shape and rank along a new axis given by a fractional number. Between 0 and 1 indicates the new axis is to be the first. Between the last present axis and one greater means the new axis is to be the last. Between the indices to any two of the present dimensions means the new axis will be inserted there. The result will have rank one greater than the arguments, and the same shape except for the new axis dimension, which is always 2.

A scalar or one-element vector as an argument is extended to conform to the shape of the other argument.

In classic APL, non-empty numeric and character arguments couldn't be laminated with one another. In APL2 and CAPLIB2 this is no longer a restriction.

APL Usage

```
apl> L # m # 3 3 R 'abcdefghi'

abc
def
ghi

apl> m,[1.5]'x'

abc
xxx

def
xxx

ghi
xxx

apl> a#'for'
apl> b#'axe'
apl> L#z#a,[.5]b

for
axe

apl> Rz

2 3

apl> ‘=z
```

```
apl> m,[.5]'x'

abc
def
ghi

xxx
xxx
xxx

apl> m,[2.5]'x'

ax
bx
cx

dx
ex
fx

gx
hx
ix

apl> L#w#a,[1.1]b

fa
ox
re

apl> Rw

3 2
```

```
1

apl> h#(1 2) (3 4)
apl> k#'ab' 'cd'
apl> L#y#h,[.5]k

 1 2  3 4
  ab   cd

apl> Ry

2 2

apl> ‘=y

2

apl> L#q#3 3R'styhimred'

sty
him
red

apl> 'a',[1.1]q

aaa
sty

aaa
him

aaa
red
```

```
apl> ‘=w

1

apl> L#v#h,[1.1]k

 1 2  ab
 3 4  cd

apl> Rv

2 2

apl> ‘=v

2

apl> 'a',[.1]q

aaa
aaa
aaa

sty
him
red

apl> 'a',[2.1]q

as
at
ay

ah
ai
am

ar
ae
ad
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Laminate; Litvect; Perm;
     Quadout; Ravel; Reshape; Temp;
     Aplcb m,n;
     #include "aplinit.h"

     m=perm(
          quadout(
               reshape(
                    litvect("3 3"),
                    chrvect("ABCDEFGHI")
               )
          )
     );
/*                                    ABC
                                      DEF
                                      GHI
*/
     n=perm(
          quadout(
               reshape(
                    litvect("3 3"),
                    chrvect("123456789")
               )
          )
     );
/*
                                     123
                                     456
                                     789
*/
     endoper(quadout(laminate(m,n,0.5)));
/*
                                     ABC
                                     DEF
                                     GHI
```

```
                                        123
                                        456
                                        789
*/
     endoper(quadout(laminate(m,n,1.5)));
/*
                                        ABC
                                        123

                                        DEF
                                        456

                                        GHI
                                        789
*/
     endoper(quadout(laminate(m,n,2.5)));
/*
                                        A1
                                        B2
                                        C3

                                        D4
                                        E5
                                        F6

                                        G7
                                        H8
                                        I9
*/
     endoper(temp(m));
     endoper(temp(n));
     endoper(
          quadout(
               ravel(
                    laminate(
                         reshape(
                              litvect("2 2"),
                              chrvect("1234")
                         ),
                         chrvect("+"),
                         2.5
                    )
```

```
                )
            )
        );
/*                                          1+2+3+4+ */
}
```

See Also

Catenate (,).

LATENT EXPRESSION ⎕LX **Llx**

Quad lx is a system variable to which you assign a character string containing an APL expression to be executed automatically whenever the workspace is loaded. If **Llx** exists in that workspace and is non-blank, APL automatically passes it as argument to the **execute** function.

The **help** workspace that accompanies CAPLIB^2 is an example of how `Llx` might be used. This workspace contains a function called **help** that offers one-line messages describing either a particular key on the keyboard, or an APL function such as **shape** or **reduction**. Here's an example of calling help to find out what the slash key (/) does:

```
apl> help '/'

/   slash symbol, compress and reduction
/   compress, 1 0 1/I3 is 1 3
/   reduction, +/I3 is 6
```

And a second example, finding out about **scan**:

```
apl> help 'scan'

\   scan +\1 2 3 is 1 3 6
‘\  scan, +‘\2 3RI6 is 2 3R1 2 3 5 7 9
```

I wanted to set this workspace up so that if you happened to load it as the active workspace it would automatically give you a message telling you what it's for. So I did the following in APL:

```
apl> )load help
apl> msg1#Q
apl> entering e.g. "help 'R'" tells you what the R key does.
apl> msg2#Q
apl> entering e.g. "help 'reshape'" tells you about reshape.
apl> cols#(Rmsg1)SRmsg2
apl> helpmsg#(colsYmsg1),[0.5]colsYmsg2
apl> Lex 3 5R'msg1 msg2 cols '
apl> Llx # 'helpmsg'
apl> )save
```

Now when you load the **help** workspace you automatically receive the following two messages:

```
    entering e.g. "help 'R'" tells you what the R key does.
    entering e.g. "help 'reshape'" tells you about reshape.
apl>
```

See Also

Appendix E, Assign (#), Expunge (Lex), Maximum (S), Reshape (R), Shape (R).

LE $\leq$ K

For real numbers, returns 1 if the left argument is less than or equal the right argument plus **comparison tolerance Lct**. Returns 0 otherwise.

For complex numbers, the results of comparing their magnitudes is returned.

APL Usage

```
apl> 1 2 3 K 3 2 1

1 1 0
```

C Usage

Subroutines **dblle, dbllep, ile, ilep, le**, and **lep** are related, differing in the data types of arguments accepted and of the value returned. Subroutine **le** accepts arguments of type **double**, and returns a value of type **int**.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dblle; Dbllep; Dyadic; Endoper;
     Idyadic; Ile; Ilep; Le; Lep;
     Litvect; Quadout; Relative;
     double left[2],rite[2],ret[2];
     int ileft,irite,iret;
     #include "aplinit.h"

     endoper(quadout(dyadic(dblle,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(relative(le,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(idyadic(ile,
          litvect("1 2 3"),
          litvect("3 2 1"))));
     *left = 2.1;
     *rite = 5.6;
     dbllep(left,rite,ret);
     printf("dbllep returned %f\n",*ret);
     lep(left,rite,&iret);
     printf("lep returned %d\n",iret);
     ileft = 2;
     irite = 5;
     ilep(&ileft,&irite,&iret);
     printf("ilep returned %d\n",iret);
}
```

```
1 1 0
1 1 0
1 1 0
dbllep returned 1.000000
lep returned 1
ilep returned 1
```

See Also

Relative Functions.

LEFT ARGUMENT □L Ll

Quad l is an APL2 system variable not implemented in CAPLIB2. In APL2, it yields the array value of the left argment of the first line of the state indicator.

LINE COUNTER □LC Llc

Quad LC contains the line numbers of defined functions that are halted, suspended or pendent, the most recently halted first.

APL Usage

```
apl> Lfx 'g' '''g line 1''' '''g: '',Llc' 'h'

g

apl> c#'h' '''h line 1''' '''h line 2'''
apl> c#c,'''h line 3''' '''h: '',Llc'
apl> Lfx c, 'j' '''h line 6'''

h

apl> c#'j' '''j line 1''' '''j line 2'''
apl> c#c,'''j line 3''' '''j line 4'''
apl> Lfx c, '''j line 5 '''  '''j: '',Llc' '''j line 7'''

j
```

```
apl> Lcr 'g'

g
'g line 1'
'g: ',Llc
h

apl> g

g line 1
 g:  2
```

```
apl> Lcr 'h'

h
'h line 1'
'h line 2'
'h line 3'
'h: ',Llc
j
'h line 6'

apl> Gj[6.1] JJJJJG
```

```
apl> Lcr 'j'

j
'j line 1'
'j line 2'
'j line 3'
'j line 4'
'j line 5'
'j: ',Llc
'j line 7'
```

```
h line 1
h line 2
h line 3
 h:  4 3
j line 1
j line 2
j line 3
j line 4
j line 5
 j:  6 5 3
j line 7
h line 6

apl> factr -6

z#!a
  ^ 36 - invalid argument to factorial.
The error occurred in function factr
factr[1]

apl> Llc

1 1

apl> g

g line 1
 g:  2 1 1
h line 1
h line 2
h line 3
 h:  4 3 1 1
j line 1
j line 2
j line 3
j line 4
j line 5
 j:  6 5 3 1 1
 JJJJJ
      ^ 70 - missing right argument during execute.
The error occurred in function j
j[7]
```

```
apl> Lfx 'z#factr a' 'z#!a'

factr

apl> factr -3

z#!a
  ^ 36 - invalid argument to factorial.
The error occurred in function factr
factr[1]

apl> Llc

1
```

```
apl> Llc

7 5 3 1 1

apl> )si

j[7] *
h[5]
g[3]
factr[1] *
factr[1] *

apl> Gj[H7]G
apl> $Llc

h line 6
```

See Also

State Indicator ()SI).

)LIB

Reports in alphabetical order, starting with the indicated letter if given, a list of the workspace names on disk in the current directory. IBM's APL2 permits a range of starting letters, as in ")LIB I-P". This is not implemented in CAPLIB2.

APL Usage

)lib [**a**]

See Also

Name List (Lnl), Appendix E.

LITVECT

C program usage only, **litvect** accepts as argument a character string containing one or more numeric constants, separated by spaces. If the constants are all syntactically correct, **litvect** converts them to internal machine representation in an array of dynamically allocated memory. **Litvect** creates and returns the address of an **aplcb** for this vector.

An **aplcb** describes an APL variable. The description for it on page 29 shows a field called **aplflags** which, among other things, describes the data type of the variable being represented. You'll see that APL variables can be either **character**, **integer**, or **double floating point**. **Litvect** knows whether the string of constants it received contained only integers, or included one or more fractional real numbers, and will construct an **aplcb** of the appropriate data type accordingly.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(litvect("1 23.45 6.789e+2")));
}
```

```
1 23.45 678.9
```

See Also

CHRVECT, CONVECT, INDXSUB.

)LOAD

)load wsname

The named workspace is loaded from disk into memory, becoming the active workspace.

Immediately after loading, the **latent expression** (Llx) just loaded is executed. This can be any valid APL expression. Here is one that reports 3, the sum of 1 and 2:

L*lx* # ‘ 1+2 ‘

Workspaces on disk under MS-DOS have the qualifier *.apl*, which you should leave off on the)load command. You can however qualify the name with its directory path, e.g. *)load \cap\handbook\wsname*.

See Also

Appendix E, Assign (#), Plus (+).

LOGARITHM, GENERAL ⍟ @

Returns the **base n** log of the right argument, the left argument being the **base n**. For any valid arguments **left** and **right**, this CAPLIB2 expression always produces 1:

```
(left @ right) ‘= (@right) % @left
```

APL Usage

```
apl> 2 4 10  @  16
4 2 1.2041199827
apl> 10@100 500 1000
2 2.69897 3
apl> 2@0j2
2.88539j2.2661805
apl> 2@256
8
apl> 1@1
1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dyadic; Endoper; Litvect; Logrithm; Quadout;
     #include "aplinit.h"

     endoper(quadout(dyadic(logrithm,
          litvect("2 4 10"),
          litvect("16 16 16"))));
}

4 2 1.2041200
```

See Also

Logarithm, Natural (@).

LOGARITHM, NATURAL ⍟ @

Returns the **natural logarithm** its argument. CAPLIB2 gives answers for complex numbers, but not for negative numbers, as IBM's APL2 will.

APL Usage

```
apl> @ 1 2 3 4

0 0.6931471806 1.0986122887 1.3862943611

apl> * @ 1 2 3 4

1 2 3 4

apl> @1

0

apl> @2.7182818284

1

apl @0j1

0j1.5707
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Log; Monadic; Quadout;
     #include "aplinit.h"

     endoper(quadout(monadic(log,
          litvect("1 2 3 4"))));
```

```
}
```

```
0 0.6931472 1.0986123 1.3862944
```

See Also

Exponential (`*`), Logarithm, General (`@`).

LT $<$ `<`

For real numbers, returns 1 if the left argument is less than the right argument minus **comparison tolerance Lct**. Returns 0 otherwise. For complex number arguments, the results of comparing their magnitudes is returned.

APL Usage

```
apl> 1 2 3 < 3 2 1

1 0 0
```

C Usage

Subroutines **dbllt, dblltp, ilt, iltp, lt,** and **ltp** are related, differing in the data types of arguments accepted and of the value returned. They all, however, implement the **less than** comparison operation.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dbllt; Dblltp; Dyadic; Endoper;
     Idyadic; Ilt; Iltp; Litvect; Lt; Ltp;
     Quadout; Relative;
     double left[2],rite[2],ret[2];
     int ileft,irite,iret;
     #include "aplinit.h"

     endoper(quadout(dyadic(dbllt,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(relative(lt,
```

```
        litvect("1.1 2.2 3.3"),
        litvect("3.3 2.2 1.1"))));
    endoper(quadout(idyadic(ilt,
        litvect("1 2 3"),
        litvect("3 2 1"))));
    *left = 2.1;
    *rite = 5.6;
    dblltp(left,rite,ret);
    printf("dblltp returned %f\n",*ret);
    ltp(left,rite,&iret);
    printf("ltp returned %d\n",iret);
    ileft = 2;
    irite = 5;
    iltp(&ileft,&irite,&iret);
    printf("iltp returned %d\n",iret);
}
```

```
1 0 0
1 0 0
1 0 0
dblltp returned 1.000000
ltp returned 1
iltp returned 1
```

See Also

Relative Functions.

MAGNITUDE | |

Monadic **magnitude**, | x, yields the absolute value of x. For any scalar **right**, including complex numbers, this CAPLIB^2 expression yields 1:

```
(|right) = (+/(9 11 0 right)*2)*.5
```

APL Usage

```
apl> | -3 0 4 -4.2
3 0 4 4.2
apl> | 2j-3
```

```
apl> | 2 -2 .3 -.3
2 2 0.3 0.3
apl> | 0j1 2j-2 4j3
```

```
3.6055513                       1   2.8284271   5
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dabs; Dabsp; Dabsx; Endoper; Iabs; Iabsp;
     Imonadic; Litvect; Monadic; Quadout;
     double rite[2],ret[2];
     int irite,iret;
     #include "aplinit.h"

     endoper(quadout(monadic(dabs,
          litvect("3.14 -3.14"))));
     *rite = -5.6;
     dabsp(rite,ret);
     printf("dabsp returned %f\n",*ret);
     *rite = 2.0;
     *(rite+1) = -3.0;
     dabsx(rite,ret);
     printf("dabsx returned %fj%f\n",*ret,*(ret+1));
     endoper(quadout(imonadic(iabs,
          litvect("3 -3"))));
     irite=-1;
     iabsp(&irite,&iret);
     printf("iabsp returned %d\n",iret);
}
```

```
3.14 3.14
dabsp returned 5.600000
dabsx returned 3.605551j0.000000
3 3
iabsp returned 1
```

MATCH ≡ ‘=

CAPLIB2 and APL2 only, does not exist in classic APL. Match yields a 1 if the arguments are the same in structure and data, and 0 otherwise.

APL Usage

```
apl> 'to' 'me' `= 'to me'        apl> 'to' 'me' `= 'to' 'me'

0                                1

apl> 1 2 3 4 `= 1 2 3 4          apl> 1 2 3 4  = 1 2 3 4

1                                1 1 1 1

apl> '' `= I0                    apl> '' `= ''

0                                1

apl> (0 2R0) `= (0 2R' ')        apl> (0 2R0) `= (0 2R0)

0                                1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Quadout; Litvect; Match;
     #include "aplinit.h"

     endoper(quadout(match(
       litvect("1 2 3"),
       litvect("1 2 3"))));
     endoper(quadout(match(
       litvect("1 2 3"),
       litvect("4 5 6"))));
}

1
0
```

MATMULT

C programming usage only, **matmult** gives the matrix product of two matrices. It declares itself as follows:

```
double *matmult(a,b,arows,acols,bcols)
double *a,*b;
int arows,acols,bcols;
```

Arguments **a** and **b** point to double floating point arrays. **Arows** and **acols** describe the number of rows and columns in the array pointed to by **a**. **Acols** also tells how many rows the array pointed to by **b** has, and **bcols** tells how many columns. **Matmult** calls **malloc** to allocate storage to contain the **matrix product** of **a** and **b**, and returns a pointer to this storage.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Matmult;
     static double a[3][3] = {
          {11.0, 12.0, 13.0},
          {50.0,  0.0, 42.0},
          {37.0, 73.0, 26.0}};
     static double b[3] = {218.0, 544.0, 805.0};
     double *answer;
     #include "aplinit.h"

     answer=matmult((double *)a,(double *)b,3,3,1);
     printf("answer[0,0]=%4.2f\n",*answer);
     printf("answer[1,0]=%4.2f\n",*(answer+1));
     printf("answer[2,0]=%4.2f\n",*(answer+2));
     free(answer);
}

answer[0,0]=19391.00
answer[1,0]=44710.00
answer[2,0]=68708.00
```

See Also

Inner Product.

MATRIX DIVIDE ⊞ M

Suppose on three occasions you bought certain quantities of three items A, B, and C, spending the total amounts given in the table below:

total	A	B	C
$4.50	3	7	1
$6.23	2	5	9
$5.19	4	3	4

This is a system of linear equations, and the solution is found with the expression given below.

APL Usage

```
5 2 F 4.50 6.23 5.19 M 3 3 R 3 7 1 2 5 9 4 3 4

    0.71 0.29 0.38
```

C Usage

APL Usage

```
apl> L#r#2 2R1 0 0 2            apl> L#l#2 2R1 2 4 8

1 0                             1 2
0 2                             4 8

apl> 1 4 M r                    apl> l M r

1 2                             1 2
                                2 4
apl> v#1 1.2 1.4 1.6 1.8 2
apl> L#l#!v

1 1.1018025 1.2421693 1.4296246 1.6764908 2


apl> 1.6 B ‘O l M v J.* 0,I2

1.4340110

apl> 1.6 B ‘O l M v J.* 0,I3

1.4289585

apl> 1.6 B ‘O l M v J.* 0,I4
```

```
1.4295805

apl> 1.6 B ‘O 1 M v J.* 0,I5

1.4296246
```

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Mdivide; Quadout; Reshape;
     #include "aplinit.h"

     endoper(quadout(mdivide(
       litvect("4.5 6.23 5.19"),
       reshape(
          litvect("3 3"),
          litvect("4 6 0 3 2 2 1 3 4")))));
}
```

```
1.5046154 -0.25307692 1.1111538
```

See Also

Format (F), Matrix Inverse (M).

MATRIX INVERSE ⌹ M

Produces the matrix inverse.

APL Usage

```
apl> M  L  #  3 3  R  4 6 0 3 2 2 1 3 4

4 6 0
3 2 2
1 3 4

-0.0384615385  0.4615384615 -0.2307692308
 0.1923076923 -0.3076923077  0.1538461538
-0.1346153846  0.1153846154  0.1923076923
```

```
apl> L#r#3 3R1 0 0 0 2 0 2 0 4   apl> L#t#Mr

1 0 0                             1.0 0.0 0.00
0 2 0                             0.0 0.5 0.00
2 0 4                            -0.5 0.0 0.25

apl> L#q#3 3 R 1 2 3 2 4 5 3 5 6

1 2 3                            apl> t +.X r
2 4 5
3 5 6                            1 0 0
                                 0 1 0
apl> L#z#Mq                      0 0 1

 1 -3  2                         apl> q +.X z
-3  3 -1
 2 -1  0                         1 0 0
                                 0 1 0
                                 0 0 1

apl> L#r#4 3 R 1 0 0 0 2 0 2 0 2 0 1 4

1 0 0                            apl> M3 4
0 2 0
2 0 2                             .12 0.16
0 1 4

apl> L#z#Mr

 0.24705882  0.09411765  0.37647059 -0.18823529
 0.04705882  0.49411765 -0.02352941  0.01176471
-0.05882353 -0.11764706  0.02941176  0.23529412

apl> z +.X r                     apl> M3

1 0 0                             .33333333
0 1 0
0 0 1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Matinv; Monadic;
     Quadout; Reshape;
     #include "aplinit.h"

     endoper(quadout(matinv(reshape(
          litvect("3 3"),
          litvect("4 6 0 3 2 2 1 3 4")))));
}
```

```
-0.03846154  0.46153846 -0.23076923
 0.19230769 -0.30769231  0.15384615
-0.13461538  0.11538462  0.19230769
```

See Also

Matrix Divide (M).

MAXIMUM ⌈ S

Returns the greater of the left and right arguments.

APL Usage

```
apl> 1.1 2.3 5  S  3.1 2.1 1.1

3.1 2.3 5
```

C Usage

Subroutines **dblmax, dblmaxp, imax,** and **imaxp** are related, differing in the data types of arguments accepted and of the value returned.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
```

```
     Dyadic; Dblmax; Dblmaxp; Endoper;
     Idyadic; Imax; Imaxp; Litvect; Quadout;
     double left,rite,ret;
     int ileft,irite,iret;
     #include "aplinit.h"

     endoper(quadout(dyadic(dblmax,
          litvect("1.1 2.3 5"),
          litvect("3.1 2.1 1.1"))));
     endoper(quadout(idyadic(imax,
          litvect("1 2 5"),
          litvect("3 2 1"))));
left=1.0;
rite=2.0;
     dblmaxp(&left,&rite,&ret);
     printf("dblmaxp returned %f\n",ret);
     ileft=1;
     irite=2;
     imaxp(&ileft,&irite,&iret);
     printf("imaxp returned %d\n",iret);
}

3.1 2.3 5
3 2 5
dblmaxp returned 2.000000
imaxp returned 2
```

See Also

Minimum (D).

MEMBERSHIP ε E

Membership, v E s, yields an array of 1s and 0s the same shape as **v**, an element being 1 if the corresponding element of **v** exists somewhere in **s**.

APL Usage

```
apl> 'abcdefgh' E 'coffee'          apl> 'banana' E 'an'

0 0 1 0 1 1 0 0                     0 1 1 1 1 1
```

```
apl> 5 1 2 E 6 5 4 1 9

1 1 0

apl> L # a # 2 3 R 8 3 5 8 4 8

8 3 5
8 4 8

apl> L # b # 'ah' 'ha' 'ah' 'no'

 ah ha ah no

apl> R b

4

apl> b E `Z 'ah'

1 0 1 0

apl> Rc

3

apl> c E (1 2) (3 5) (I0)

1 1 0
```

```
apl> a E 1 8 9 3

1 1 0
1 0 1

apl> b E 'ah'

0 0 0 0

apl> `= b

2

apl> L#c#(1 2) (I0) (3 4)

 1 2    3 4

apl> `=c

2

apl> 8 9 7 3 E I0

0 0 0 0
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Member; Quadout;
     #include "aplinit.h"

     endoper(quadout(member(
       chrvect("abcdefgh"),
       chrvect("coffee"))));
```

```
}
```

0 0 1 0 1 1 0 0

See Also

Compress (/), Expand (\ ), Index Of (I).

MINIMUM ⌊ D

Returns the lesser of the left and right arguments.

APL Usage

```
apl> 1.1 2.3 -5 D 3.1 2.1 1.1

1.1 2.1 -5

apl> 5 D 4 5 7

4 5 5

apl> 3.3 0 -6.7 D 3.1 -4 -5

3.1 -4 -6.7
```

```
apl> 3 D 4

3

apl> -2 D -3

-3
```

C Usage

Subroutines **dblmin, dblminp, imin,** and **iminp** are related, differing in the data types of arguments accepted and of the value returned. **Min** accepts arguments of type **double**, and returns a value of type **double**.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dblmin; Dblminp; Dyadic; Endoper;
     Idyadic; Imin; Iminp; Litvect; Quadout;
     double left,rite,ret;
     int ileft,irite,iret;
     #include "aplinit.h"
```

```
     endoper(quadout(dyadic(dblmin,
          litvect("1.1 2.3 -5"),
          litvect("3.1 2.1 1.1"))));
     endoper(quadout(idyadic(imin,
          litvect("1 2 5"),
          litvect("3 2 0"))));
left=1.0;
rite=2.0;
     dblminp(&left,&rite,&ret);
     printf("dblminp returned %f\n",ret);
     ileft=1;
     irite=2;
     iminp(&ileft,&irite,&iret);
     printf("iminp returned %d\n",iret);
}
```

```
1.1 2.1 -5
1 2 0
dblminp returned 1.000000
iminp returned 1
```

See Also

IMAX, IMIN, Maximum (S).

MINUS − _

Returns the standard arithmetic difference of its left and right arguments.

APL Usage

```
apl> -3.1 0 3.1 _ -3.2 0 3.2

0.1 0 -0.1

apl> 6 _ 8 .2 4j3

-2   5.8   2j-3

apl> -4 .5 0 _ -2 1.2 1j2
```

```
apl> 5 _ 3

2

apl> 3j4 _ 1j2

2j2
```

```
-2   -0.7   -1j-2
```

C Usage

Subroutines **iminus** and **minus** are related, differing in the data types of arguments accepted and of the value returned. **Minus** accepts arguments of type **double**, and returns a value of type **double**.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dyadic; Endoper; Idyadic; Iminus; Litvect;
     Minus; Minusp; Minusx; Quadout;
     double left[2],rite[2],ret[2];
     #include "aplinit.h"

     endoper(quadout(dyadic(minus,
          litvect("-3.1 0 3.1"),
          litvect("-3.2 0 3.2 "))));
     endoper(quadout(idyadic(iminus,
          litvect("-3 -2 -1 0  1  2 3"),
          litvect("-3  2  1 0 -1 -2 3"))));
     *left = 2.1;
     *rite = 5.6;
     minusp(left,rite,ret);
     printf("minusp returned %f\n",*ret);
     *left = 0;
     *(left + 1) = 12.0;
     *rite = 4.0;
     *(rite + 1) = 0;
     minusx(left,rite,ret);
     printf("minusx returned %fj%f\n",*ret,*(ret+1));
}
```

```
 .1 0 -0.1
0 -4 -2 0 2 4 0
minusp returned -3.500000
minusx returned -4.000000j12.000000
```

See Also

IMINUS

MONADIC

Monadic is both the name of a CAPLIB^2 routine and an attribute of certain APL primitive functions and operators, which are said to be **monadic** if they accept one argument.

The CAPLIB^2 routine named **monadic** extends monadic scalar functions such as **factorial (!)** and **absolute value (|)** to an APL variable containing double floating point data item-by-item. If the APL variable isn't double floating point, **monadic** will do the necessary conversion.

See Also

Numerous examples of the use of **monadic** exist throughout this manual. See, for example, Ceiling (S), Factorial (!). See also IMONADIC, DYADIC, and NILADIC.

NAME ASSOCIATION ⎕NA **Lna**

Quad na is an APL2 system variable not implemented in CAPLIB^2.

NAME CLASSIFICATION ⎕NC **Lnc**

Quad nc (Lnc) is a monadic system function that accepts either a character vector containing a single name, or a matrix containing a name in each row. For each name, **Lnc** returns an integer 0 thru 4 as follows:

0 - Name is available for any use.

1 - Name is a label.

2 - Name is a variable.

3 - Name is a defined function.

4 - Argument is not available for use as a name, perhaps because it isn't a valid name, i.e."123abc".

Quad nc in CAPLIB^2 differs slightly from that in IBM's APL2, which returns -1 to indicate an invalid name, and 4 to indicate an operator.

APL Usage

```
apl> nickname # 'smitty'
apl> vec # I5
apl> L # names # 4 8 R'nicknamevec     hello   123abc  '

nickname                        apl> Lnc names
vec
hello                           2 2 0 4
123abc

apl> Ghello                     apl> Lnc names
[1]    'hello world'
[2]    G                        2 2 3 4
```

C Usage

See the example for **expunge** beginning on page 131.

See Also

Appendix E, Assign (#), Function Editing (G), Name List (Lnl), Reshape (R).

NAME LIST □NL Lnl

Quad nl (Lnl) is a system function that returns a character matrix, each row of which is the name of an object in the active workspace, either label, variable, or defined function. It accepts as its right argument a scalar or vector of numbers that determines the classes of names to be returned, as follows:

1 - Labels.

2 - Variables.

3 - Defined functions.

IBM's APL2 also returns a 4, to indicate a defined operator. This is not implemented in C$_{\text{APL}}$IB^{2}.

Quad nl also accepts a left argument, which must be a scalar or vector of alphabetic characters. The names produced will be restricted to those having their initial letter as a member of this argument.

APL Usage

```
apl> nickname # 'smitty'
apl> vec # I5
apl> Ghello
[1]    'hello world'
[2]    G

apl> Lnl 2                       apl> Lnl 3

nickname                         hello
vec

apl> Lnl 1 2 3                   apl> 'hn' Lnl 1 2 3

hello                            hello
nickname                         nickname
vec
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Execute;
     Execqnl; Litvect; Quadout;
     #include "aplinit.h"

     endoper(execute(chrvect("aaa # I5")));
     endoper(execute(chrvect("bbb # 'abc'")));
     endoper(execute(chrvect("bcd # 'whatever'")));
     endoper(quadout(execqnl(NULL,litvect("1 2 3"))));
     endoper(quadout(execqnl(
       chrvect("b"),
       litvect("1 2 3"))));
}

aaa
bbb
bcd
```

```
bbb
bcd
```

See Also

Appendix E, Assign (#), Index Generator (I), Name Classification (Lnc).

NATIONAL LANGUAGE TRANSLATION ⎕NLT **Lnlt**

Quad nlt is an APL2 system variable not implemented in CAPLIB2.

NAMES

You name **workspaces, defined functions, variables**, and **groups** in APL. The rules for all but workspaces are the same. Workspace names become DOS *filenames*, so their formation must comply with DOS rules, and they must be from one to eight characters in length.

The rule for names other than workspaces is that they may consist of any sequence of alphabetic and numeric characters that starts with an alphabetic and contains no blank. Theoretically, such names can be any length you like, but practically you may want to limit yourself to something less than forty or fifty characters.

Alphabetics in the CAPLIB2 environment consist of any of the lowercase letters **a** thru **z**, the accented letters **'a** thru **'z**, and the **delta** symbol (H). Numerics of course consist of the digits 0 thru 9. Appendix B beginning on page 349 shows the ASCII codes for these symbols.

Local and Global Names

Names for arguments and the result variable in the template on the function header are said to be **local** to the defined function, as opposed to being **global** (**Defined functions** and **function headers** are discussed beginning on page 86). Here are some rules which describe the differences:

- Instances of variables with localized names exist only from their creation until the termination of the execution of the defined function in which they're localized.

- As long as a name is localized, a global variable with the same name is **shadowed**, i.e. it can't be referenced or changed. This shadowing is extended to any defined functions invoked by the current defined function. If a defined function attempts to reference or change a variable with a name which is not localized to itself, but which is localized in some calling defined function, that is the instance that will be referenced or changed.
- If a defined function makes an assignment to a variable with a name not localized either to itself or to any calling function, a new instance is created which is global to all.

See Also

Appendix J, DEFINED FUNCTION,)SAVE.

NOT EQUAL $\neq$ $\wedge =$

Returns 1 if the difference of the absolute values of the left and right arguments is greater than *comparison tolerance*, or the signs of the two arguments differ. Returns 0 otherwise.

APL Usage

```
apl> 1 2 3 ^= 3 2 1

1 0 1
```

C Usage

Subroutnes **dblne, dblnep, ine, ne**, and **nep** are related, differing in the data types of arguments accepted and of the value returned.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dblne; Dblnep; Dyadic; Endoper;
     Idyadic; Ine; Litvect;
     Ne; Nep; Relative; Quadout;
     double left[2],rite[2],ret[2];
     int ileft,irite,iret;
```

```
     #include "aplinit.h"

     endoper(quadout(dyadic(dblne,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(relative(ne,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(idyadic(ine,
          litvect("1 2 3"),
          litvect("3 2 1"))));
     *left = 2.1;
     *rite = 5.6;
     dblnep(left,rite,ret);
     printf("dblnep returned %f\n",*ret);
     nep(left,rite,&iret);
     printf("nep returned %d\n",iret);
}
```

```
1 0 1
1 0 1
1 0 1
dblnep returned 1.000000
nep returned 1
```

See Also

Relational functions.

NEGATIVE – _

Negative returns its argument reversed in sign.

APL Usage

```
apl> _ -12.3 45.6              apl> _0j1 -3j4 -2j-1

12.3 -45.6                      0j-1 3j-4 2j1
```

C Usage

Subroutines **inegatip, inegativ, negative,** and **negativp** are related, differing in the data types of arguments accepted, and the value returned.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Imonadic; Inegatip; Inegativ;
     Litvect; Monadic; Negative; Negativp;
     Negativx; Quadout;
     double rite[2],ret[2];
     int irite,iret;
     #include "aplinit.h"

     endoper(quadout(monadic(negative,
          litvect("-12.3 45.6"))));
     endoper(quadout(imonadic(inegativ,
          litvect("-123 456"))));
     *rite = -5.6;
     negativp(rite,ret);
     printf("negativp returned %f\n",*ret);
     *rite = 1.0;
     *(rite+1) = -2.0;
     negativx(rite,ret);
     printf("negativx returned %fj%f\n",*ret,*(ret+1));
     irite=-1;
     inegatip(&irite,&iret);
     printf("inegatip returned %d\n",iret);
}

12.3 -45.6
123 -456
negativp returned 5.600000
negativx returned -1.000000j2.000000
inegatip returned 1
```

See Also

INEGATIV.

NEGATIVE NUMBERS

In C, the prefix minus indicating a negative number, and the infix minus indicating subtraction are one and the same, i.e. −. But in APL an **overbar** ("¯" in the APL symbol set, and "−" in the CAPLIB2 symbol set) is prefixed to a number to indicate it is negative, and the bar ("−" in the APL symbol set, and "_" in the CAPLIB2 symbol set) is used between operands to indicate subtraction.

APL Usage

```
apl> " This indicates subtraction of two positive numbers.
apl> 5 _ 2

3

apl> " This indicates addition of two negative numbers.
apl> -3 + -5

-8
```

See Also

Constants, Minus (_), Plus (+)

NILADIC

Niladic is an attribute of certain user-defined functions which accept no arguments.

See Also

Defined Functions, DYADIC, and MONADIC.

NOR ⍱ ‘V

Returns 1 if neither of the absolute values of the left and right arguments is greater than or equal to **comparison tolerance Lct**. Returns 0 otherwise.

APL Usage

```
apl> 1 2 3 0 0 1 1  ‘V  3 2 1 0 1 0 1

0 0 0 1 0 0 0
```

C Usage

Subroutines **dblnor, dblnorp, inor, inorp, nor**, and **norp** are related, differing in the data types of arguments accepted and of the value returned.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dblnor; Dblnorp; Dyadic; Endoper;
     Idyadic; Inor; Inorp; Litvect;
     Nor; Norp; Relative; Quadout;
     double left=1.0,rite=1.0,ret;
     int ileft=1,irite=1,iret;
     #include "aplinit.h"

     endoper(quadout(relative(nor,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(dyadic(dblnor,
          litvect("0.0 0.0 1.0 1.0"),
          litvect("0.0 1.0 0.0 1.0"))));
     endoper(quadout(idyadic(inor,
          litvect("1 2 3"),
          litvect("3 2 1"))));
     norp(&left,&rite,&iret);
     printf("norp returned %d\n",iret);
     dblnorp(&left,&rite,&ret);
     printf("dblnorp returned %f\n",ret);
     printf("inor returned %d\n",inor(ileft,irite));
     inorp(&ileft,&irite,&iret);
     printf("inorp returned %d\n",iret);
}

0 0 0
1 0 0 0
0 0 0
```

```
norp returned 0
dblnorp returned 0.000000
inor returned 0
inorp returned 0
```

See Also

Comparison Tolerance (Lct), INOR, DBLNOR.

NOT ~ ~

Returns 1 if the absolute value of the argument is less than **comparison tolerance Lct**. Returns 0 otherwise.

APL Usage

```
apl> ~ 3 2 1 0

0 0 0 1
```

C Usage

Subroutines **dblnot, dblnotp, inot**, and **not** are related, differing in the data types of arguments accepted, and of the value returned. They all, however, implement the **not** logical operation.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dblnot; Dblnotp; Endoper; Imonadic; Inot;
     Litvect; Monadic; Not; Quadout;
     double rite,ret;
     #include "aplinit.h"

     endoper(quadout(monadic(dblnot,
          litvect("3.0 2 1 0"))));
     endoper(quadout(imonadic(inot,
          litvect("3 2 1 0"))));
     rite = -5.6;
     dblnotp(&rite,&ret);
     printf("dblnotp returned %f\n",ret);
```

```
     rite = 1.0;
     printf("not returned %d\n",not(rite));
}

0 0 0 1
0 0 0 1
dblnotp returned 0.000000
not returned 0
```

See Also

Comparison Tolerance (⎕ct), INOT, DBLNOT.

)OFF

)off

Terminates the APL interpreter and returns you to a DOS prompt.

See Also

Appendix E.

OR ∨ V

Returns 1 if the absolute value of either the left or right argument is greater than *comparison tolerance*.

APL Usage

```
apl> 1 2 3 0 0 1 1 V 3 2 1 0 1 0 1

1 1 1 0 1 1 1
```

C Usage

Subroutines **dblor, dblorp, ior, iorp, or**, and **orp** are related, differing in the data types of arguments accepted, and of the value returned.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dblor; Dblorp; Dyadic; Endoper; Idyadic;
     Ior; Iorp; Litvect; Or; Orp;
     Quadout; Relative;
     double left=1.0,rite=1.0,ret;
     int ileft=1,irite=1,iret;
     #include "aplinit.h"

     endoper(quadout(relative(or,
          litvect("1.1 2.2 3.3"),
          litvect("3.3 2.2 1.1"))));
     endoper(quadout(dyadic(dblor,
          litvect("0.0 0.0 1.0 1.0"),
          litvect("0.0 1.0 0.0 1.0"))));
     endoper(quadout(idyadic(ior,
          litvect("1 2 3"),
          litvect("3 2 1"))));
     orp(&left,&rite,&iret);
     printf("orp returned %d\n",iret);
     dblorp(&left,&rite,&ret);
     printf("dblorp returned %f\n",ret);
     printf("ior returned %d\n",ior(ileft,irite));
     iorp(&ileft,&irite,&iret);
     printf("iorp returned %d\n",iret);
}
```

```
1 1 1
0 1 1 1
1 1 1
orp returned 1
dblorp returned 1.000000
ior returned 1
iorp returned 1
```

See Also

Comparison Tolerance (Lct), IOR, DBLOR.

OUTER PRODUCT ∘. J.

Outer product, aJ.fb, applies any dyadic function **f** to every element of left argument **a** paired with each member of right argument **b**. The shape of the result is the catenation of the shapes of the two arguments, **(Ra),Rb**.

Either argument of **outer product** may be a scalar or one-element vector, in which case it is extended as required by the shape of the other argument, and the result will be the same as though the scalar function had been directly applied.

APL Usage

```
apl> 2 3  J.+  4 5 6

6 7 8
7 8 9
```

```
apl> 10 20 J., 1 2 3

 10 1  10 2  10 3
 20 1  20 2  20 3
```

```
apl> 2 3 4  J.X  2

4 6 8
```

```
apl> 2 3 4  X  2

4 6 8
```

```
apl> z#'-',[1](10 1RI10),' ',' ','|',' ',z#F(I10)J.XI10
apl> ((1,-1YRz)R' X  |',((18R3 0),4 0)FI10),[1]z

    X  |  1  2  3  4  5  6  7  8  9  1
  - --------------------------------
  1    |  1  2  3  4  5  6  7  8  9  10
  2    |  2  4  6  8 10 12 14 16 18  20
  3    |  3  6  9 12 15 18 21 24 27  30
  4    |  4  8 12 16 20 24 28 32 36  40
  5    |  5 10 15 20 25 30 35 40 45  50
  6    |  6 12 18 24 30 36 42 48 54  60
  7    |  7 14 21 28 35 42 49 56 63  70
  8    |  8 16 24 32 40 48 56 64 72  80
  9    |  9 18 27 36 45 54 63 72 81  90
 10    | 10 20 30 40 50 60 70 80 90 100
```

```
apl> (I4)J.+I5

2 3 4 5 6
3 4 5 6 7
```

```
apl> (I4)J.=I4

1 0 0 0
0 1 0 0
```

```
4 5 6 7 8                        0 0 1 0
5 6 7 8 9                        0 0 0 1

apl> L # z # 3 4 J.Y 'LLLL' 'HHHHH'

 LLL  HHH                        apl> s#3 4R'theywanttrain'
 LLLL HHHH                       apl> 'at'J.=s

apl> Rz                          0 0 0 0
                                 0 1 0 0
2 2                              0 0 1 0

apl> ‘=z                         1 0 0 0
                                 0 0 0 1
2                                1 0 0 0
```

C Usage

CAPLIB2 subroutine **outrprd** declares itself in this fashion:

```
struct aplcb *outrprd(oper,left,rite)
void (*opera)();
struct aplcb *left,*rite;
```

Oper is the scalar dyadic function, such as **plus** or **times**. **Left** and **rite** are pointers to the two APL variables. All three data types are acceptable, according to the following rules:

- If either of the arguments **left** or **rite** points to an **aplcb** for character data, both must be for character, and the only two acceptable scalar dyadic functions are **ceq** and **cne**, which compare character data for **equal** and **not equal**, respectively. The result will have data type integer, consisting exclusively of the boolean values 0 and 1.

- If either of the arguments **left** or **rite** points to an **aplcb** for double floating point data, the other will be converted to double floating point data, and the result will also be double floating point. This restricts **oper** to be one of the scalar dyadic functions that accepts double floating point arguments, and returns a double floating point result, e.g. plus, minus, dbland, etc.

- If both the arguments **left** and **rite** point to **aplcb**s for integer data, the result will also be integer, restricting **oper** to one of the scalar dyadic functions that accepts integer and returns integer, e.g. iplus, iminus, iand, etc.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Ilt; Indxsub; Itimes; Outrprd; Quadout;
     #include "aplinit.h"

     endoper( /* ----------------------> 1 2 3  4  5 */
          quadout(                  /* 2 4 6  8 10 */
               outrprd(             /* 3 6 9 12 15 */
                    itimes,
                    indxsub(3),
                    indxsub(5)
               )
          )
     );
     endoper( /* ---------------------> 0 1 1 1 1 */
          quadout(                 /* 0 0 1 1 1 */
               outrprd(            /* 0 0 0 1 1 */
                    ilt,
                    indxsub(3),
                    indxsub(5)
               )
          )
     );
}
```

See Also

Inner Product (.), Plus (+), Times (X).

PARENTHESES () ()

In standard mathematics notation and many programming languages like C, PL/1, and BASIC, the order of evaluation in arithmetic expressions is from left to right, except some operators take precedence over others. For example, $3 + 2 \times 4$ evaluates to 11, not 24. In APL, however, expressions are evaluated right to left, and no operator takes precedence over another. So in APL, **3 + 2 X 4** would yield 11, but **2 X 4 + 3** yields 24.

Like standard mathematics notation, C, PL/1, and BASIC, APL lets you override its natural order of evaluation with parentheses. Tbus, the APL expressions `(1_2_3_4_5)` and `((((1_2)_3)_4)_5)` respectively yield 3 and −13.

PARTITION ⊂ ‘Z

CAPLIB2 and APL2 function only, did not exist in classic APL. This function "partitions" its right argument according to break points specified by its left argument. Each partition becomes a new array nested in the result. The depth of the result is always 1 greater than that of the right argument.

The left argument is a simple vector (or scalar, replicated into a vector of length matching the chosen axis (see below)) of nonnegative integers. One partition ends and another begins whenever an item of the left argument is greater then the previous. If an item from the left argument is 0 the corresponding item from the right argument isn't included in the result.

A bracketed axis value may be specified, the default being the last axis of right. The length of the left argument must match the length of this axis.

These statements all produce 1:

- For any boolen vector or scalar **b**:

```
(b/r) ‘= ‘X,/b‘Zr
(b/[x]r) ‘= ‘X[x],/[x]b‘Z[x]r
```

- For q containing no zeros:

```
r ‘= ‘X,/q‘Zr
r ‘= ‘X[x],/[x]q‘Z[x]r
```

- For any appropriate **q** and **r**:

```
((0^=q)/r) ‘= ‘X,/q‘Zr
((0^=q)/[x]r) ‘= ‘X[x],/[x]q‘Z[x]r
```

APL Usage

```
apl> display 1 1 2‘Z’abc’        apl> L#z#1 1 2‘Z’abc’
```

```
+$---------+
| +$-+ +$+ |
| |ab| |c| |
| +--+ +-+ |
+E---------+

apl> (‘=z),Rz

2 2

apl> (‘=z),Rz

2 2
```

```
 ab c

apl> L#z#1 0 1‘Z’abc’

 a c

apl> display z

+$--------+
| +$+ +$+ |
| |a| |c| |
| +-+ +-+ |
+E--------+
```

```
apl> " Lengths of arguments must match
apl> L # z # 1 0 1 ‘Z ’abcd’

L # z # 1 0 1 ‘Z ’abcd’
              ^ 128 - length error.

apl> (‘=z),Rz

2 2

apl> display z

+$--------+
| +$+ +$+ |
| |a| |c| |
| +-+ +-+ |
+E--------+

apl> " Partition numeric vector into pieces
apl> L # z # 2 1 2 ‘Z 10 20 30

 10 20  30

apl> (‘=z),Rz

2 2

apl> display z
```

```
+$-------------+
| +$----+ +$-+ |
| |10 20| |30| |
| +~----+ +~-+ |
+E-------------+

apl> " Partition adds a level of nesting
apl> L # z # 'one' 'two' 'buckle my shoe'

 one two buckle my shoe

apl> (`=z),Rz

2 3

apl> L # z # 1 1 2 `Z z

  one two    buckle my shoe

apl> (`=z),Rz

3 2

apl> display z

+$-------------------------------------+
| +$------------+ +$-----------------+ |
| | +$--+ +$--+ | | +$-------------+ | |
| | |one| |two| | | |buckle my shoe| | |
| | +---+ +---+ | | +--------------+ | |
| +E------------+ +E-----------------+ |
+E-------------------------------------+

apl> " Examples with blank delimiters
apl> L # x # '  a stitch  in time   '

  a stitch  in time

apl> (`=x),Rx

1 22
```

```
apl> " Partition and discard blank delimiters
apl> L # z # (' ' ^= x) `Z x

 a stitch in time

apl> (`=z),Rz

2 4

apl> display z

+$-------------------------+
| +$+ +$-----+ +$-+ +$---+ |
| |a| |stitch| |in| |time| |
| +-+ +------+ +--+ +----+ |
+E-------------------------+

apl> " Keep delimiters on the ends
apl> L # z # (1 + ' ' ^= x) `Z x

    a  stitch   in  time

apl> (`=z),Rz

2 5

apl> display z

+$------------------------------------+
| +$-+ +$-+ +$-------+ +$--+ +$------+ |
| |  | |a | |stitch  | |in | |time   | |
| +--+ +--+ +--------+ +---+ +-------+ |
+E------------------------------------+

apl> " Keep delimiters on the beginnings
apl> L # z # (1 + ~ ' ' ^= x) `Z x

   a  stitch   in  time

apl> (`=z),Rz
```

```
2 5

apl> display z

+$------------------------------------+
| +$--+ +$------+ +$---+ +$----+ +$--+ |
| |  a| | stitch| |  in| | time| |   | |
| +---+ +-------+ +----+ +-----+ +---+ |
+E------------------------------------+

apl> " Partition a matrix at blank columns
apl> L # m # 3 12 R '1   10 3.1422  100 6.2833 1000 9.425'

1   10 3.142
2  100 6.283
3 1000 9.425

apl> L # z # (~ & `/ ' ' = m) `Z m

 1   10 3.142
 2  100 6.283
 3 1000 9.425

apl> display z

+$-------------------+
U +$+ +$---+ +$----+ |
| |1| |  10| |3.142| |
| +-+ +----+ +-----+ |
| +$+ +$---+ +$----+ |
| |2| | 100| |6.283| |
| +-+ +----+ +-----+ |
| +$+ +$---+ +$----+ |
| |3| |1000| |9.425| |
| +-+ +----+ +-----+ |
+E-------------------+

apl> " Partition with Axis
apl> display n # 4 3 R I 12

+$-------+
U 1  2  3|
```

```
| 4  5  6|
| 7  8  9|
|10 11 12|
+~-------+

apl> display 1 0 1 1 ‘Z[1] n

+$---------------------+
U +$+    +$+    +$+    |
| |1|    |2|    |3|    |
| +~+    +~+    +~+    |
| +$---+ +$---+ +$---+ |
| |7 10| |8 11| |9 12| |
| +~---+ +~---+ +~---+ |
+E---------------------+
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Quadout; Indxsub; Litvect; Partitn; Reshape;
     #include "aplinit.h"

     endoper(quadout(partitn(
          litvect("1 0 1 1"), /* left */
          reshape(litvect("4 3"),indxsub(12)), /* right */
          litvect("1")  /* axis */
     )));
}

    1     2     3
 7 10  8 11  9 12
```

See Also

Enclose (‘X).

)PCOPY

)pcopy wsname obj1 obj2 ···

The indicated objects (functions, variables, etc.) are copied from the workspace named "*wsname*." into the active workspace, except if a named object already exists in the active workspace, it won't be replaced. Other than this constraint, **)pcopy** works exactly like **)copy**.

See Also

Appendix E

PERM

C program usage only, marks an apl variable permanent. CAPLIB2 subroutine **perm** declares itself as follows:

```
struct aplcb *perm(rite)
struct aplcb *rite;
```

Aplcbs are used to represent APL variables. The complete description of an **aplcb** begins on page 29. Among the fields it contains is one called **aplflags**, which, among other things, can be set to designate the **aplcb** as being temporary. Subroutine **perm** ensures the temporary flag is turned off in the **aplcb** pointed to by its argument, then returns its argument to its caller.

Subroutines and functions described in this manual, such as DYADIC, MONADIC, QUADOUT, etc., that accept pointers to APLCBs as arguments, will call ENDOPER after they've completed their processing. Subroutines and functions that create APLCBs and return pointers to them, such as CHRVECT, CONVECT, and LITVECT, unless otherwise indicated in this manual, will mark them temporary. Thus, these APLCBs will be freed automatically the first time they are passed as an argument to DYADIC, MONADIC, etc., unless they are first marked permanent by passing them to **perm**.

See also

APLCB, ENDOPER, GETCB, TEMP.

PICK ⊃ ‘X

CAPLIB2 and APL2 function only, did not exist in classic APL. **Pick** lets you select any item at any depth from an array. These statements hold true:

- If the left argument to pick is a scalar or one-item vector, the item is selected from the outermost structure of the right argument.
- The shape of the left argument can be no greater than the depth of the right argument. Successive items of the left argument apply to deeper elements in the structure of the right argument.
- If the right argument is multidimensional, the first item of the left argument must be a vector whose length is the rank of the right argument.
- If the left argument is empty, selection is from the right argument at depth 0, and therefore, all of the right argument is selected.

APL Usage

```
apl> L#r#'four' 'to' 'go'

 four to go

apl> L#z#2 ‘X r

to

apl> Lio # 0
apl> 1 ‘X r

to

apl> Lio # 1
apl> L#a#'s' 'si' ('sir' 'sire')

 s si   sir sire

apl> L#w#2 ‘X a

si
```

```
apl> ‘=r

2

apl> (Rz),‘=z

2 1
```

```
apl> ‘=a

3

apl> (Rw),‘=w
```

```
apl> L#x#3 ‘X a                  2 1

 sir sire                        apl> (Rx),‘=x

                                 2 2

apl> L#c#2 2R’one’ ’two’ ’buckle’ (’my’ ’shoe’)

 one           two               apl> (Rc),‘=c
 buckle   my shoe
                                 2 2 3

apl> display c                   apl> L#y#(‘Z 2 2) ‘X c

+$-------------------------+      my shoe
U +$--+    +$--+           |
| |one|    |two|           |     apl> (Ry),‘=y
| +---+    +---+           |
| +$-----+ +$------------+ |     2 2
| |buckle| | +$-+ +$---+ | |
| +------+ | |my| |shoe| | |
|          | +--+ +----+ | |
|          +E------------+ |
+E-------------------------+

apl> d#c,[.5]2 2 R ’three’ ’four’ ’shut’ (’the’ ’door’)
apl> display d
                                 apl> d
++$--------------------------+    one            two
UU +$--+    +$--+            |    buckle    my shoe
|| |one|    |two|            |
|| +---+    +---+            |    three         four
|| +$-----+ +$------------+  |    shut     the door
|| |buckle| | +$-+ +$---+ |  |
|| +------+ | |my| |shoe| |  |   apl> (Rd),‘=d
||          | +--+ +----+ |  |
||          +E------------+  |   2 2 2 3
||                           |
|| +$----+  +$---+           |   apl> L#q#(‘Z 2 1 2) ‘X d
|| |three|  |four|           |
|| +-----+  +----+           |   four
```

```
|| +$---+   +$-------------+ |
|| |shut|   | +$--+ +$---+ | |
|| +----+   | |the| |door| | |
||          | +---+ +----+ | |
||          +E-------------+ |
++E--------------------------+

apl> m#'b' 'ba' ('bat' 'bath')
apl> m

 b ba   bat bath

apl> `=m

3

apl> L#a#(I0)`X0 2R0

.

apl> `=a

1

apl> L#s#2 3 R 'ab' 'cd' 'ef' 'gh' 'ij' 'kl'

 ab cd ef
 gh ij kl

apl> (1 3) 2 `X s

f

apl> L#h#2 2 R 'bucks' 'twang' 'lymph' 'froze'

 bucks twang
 lymph froze

apl> `=h

2
```

```
apl> (Rq),`=q

4 1

apl> L#q#'fly' 'paper'

 fly paper

apl> 2 4 `X q

e

apl> L#n#(I0) `X m

 b ba   bat bath

apl> `=n

3
```

```
apl> display h

+$----------------+
U +$----+ +$----+ |
| |bucks| |twang| |
| +-----+ +-----+ |
| +$----+ +$----+ |
```

```
apl> L#s#(2 1) 4 ‘X h

p

apl> ‘=s

0
```

```
| |lymph| |froze| |
| +-----+ +-----+ |
+E----------------+
```

```
apl> g#’i’ ’am’ (’for’ ’apl2’)
apl> g

 i am   for apl2

apl> ‘=g

3

apl> L#t#3 2 ‘X g

apl2

apl> 3 2 1 ‘X g

a
```

```
apl> display g

+$-----------------------+
|   +$-+ +$-------------+ |
| i |am| | +$--+ +$---+ | |
| - +--+ | |for| |apl2| | |
|        | +---+ +----+ | |
|        +E-------------+ |
+E-----------------------+

apl> (Rt),‘=t

4 1
```

```
apl> L#e#2 3R’cry’ ’vox’ ’kid’ ’jab’ (2 3 R I6) (’leg’ ’nth’)

 cry     vox         kid
 jab   1 2 3    leg nth
       4 5 6

apl> display e

+$-----------------------------+
U +$--+ +$--+   +$--+          |
| |cry| |vox|   |kid|          |
| +---+ +---+   +---+          |
| +$--+ +$----+ +$------------+ |
| |jab| U1 2 3| | +$--+ +$--+ | |
| +---+ |4 5 6| | |leg| |nth| | |
|       +~----+ | +---+ +---+ | |
|               +E------------+ |
```

```
apl> (Re),‘=e

2 3 3
```

```
+E----------------------------+

apl> (2 2) (2 3) `X e            apl> L#u#(2 3) 2 `X e

6                                nth

apl> `=u                         apl> L#j#(2 3) 2 3 `X e

1                                h

apl> `=j

0

apl> k#'elm' 'tax' 'spy' 'job' 'win'
apl> L#k#2 3Rk, `Z (2 2 R 'que' 'zig' 'had' 'for')

 elm tax         spy             apl> (Rk),`=k
 job win    que zig
            had for              2 3 3

apl> display k                   apl> L#p#(2 3)(1 2)`Xk

+$---------------------------+  zig
U +$--+ +$--+ +$--+           |
| |elm| |tax| |spy|           |  apl> (Rp),`=p
| +---+ +---+ +---+           |
| +$--+ +$--+ +$------------+ |  3 1
| |job| |win| U +$--+ +$--+ | |
| +---+ +---+ | |que| |zig| | |  apl> L#i#(2 3)(1 2) 3`Xk
|             | +---+ +---+ | |
|             | +$--+ +$--+ | |  g
|             | |had| |for| | |
|             | +---+ +---+ | |  apl> (Ri),`=i
|             +E------------+ |
+E---------------------------+  0
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
```

```
{
     Chrvect; Compute; Endoper; Litvect; Quadout;
     Perm; Pick; Temp;
     Aplcb rite;
     #include "aplinit.h"

     rite=perm(compute(chrvect("'p' 'pi' ('pie' 'pier')")));
     quadout(pick(litvect("2 1"),rite,NULL));
     quadout(pick(litvect("2 1"),rite,chrvect("tr")));
     endoper(quadout(temp(rite)));
}

p
tr
 p   tr i    pie pier
```

See Also

First (Y), Selective Specification.

PI TIMES ○ **O**

The APL circle symbol (O) when used monadically results in a value pi (π) times its argument.

APL Usage

```
apl> O1 -2 (2.75*2)               apl> O3j2

3.1415927 -6.2831853 23.758294     9.4247780j6.2831853
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Monadic; Pitimes; Quadout;
     #include "aplinit.h"

     endoper(quadout(monadic(pitimes,
          litvect("1 2 3"))));
```

```
}
```

```
3.1415927 6.2831853 9.4247780
```

PLUS + +

Plus implements the standard summation arithmetic function.

APL Usage

```
apl> 1.2  +  3.4

4.6

apl> -5 + -.3 6 3j4

-5.3   1   -2j4
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dyadic; Endoper; Idyadic; Iplus; Iscalar;
     Plus; Plusp; Plusx; Quadout; Scalar;
     double left[2],rite[2],ret[2];
     #include "aplinit.h"

     endoper(quadout(dyadic(
       plus,scalar(1.2),scalar(3.4))));
     endoper(quadout(idyadic(
       iplus,iscalar(1),iscalar(1))));
     *left = 2.1;
     *rite = 5.6;
     plusp(left,rite,ret);
     printf("plusp returned %f\n",*ret);
     *left = 0;
     *(left + 1) = 12.0;
     *rite = 4.0;
     *(rite + 1) = 0;
```

```
     plusx(left,rite,ret);
     printf("plusx returned %fj%f\n",*ret,*(ret+1));
}

4.6
2
plusp returned 7.700000
plusx returned 4.000000j12.000000
```

See Also

IPLUS.

POWER ⋆ *

If **x** and **y** are its left and right arguments, **power** returns the **y**th power of **x**, according to the following two rules:

- If the absolute value of **y** is less than **comparison tolerance** (described on page 77), return 1.0 regardless of the value of **x**.
- If **y** is within **comparison tolerance** of 1, the result matches the left argument **x**.
- If the right argument **y** is a positive integer, the result is equal to the CAPLIB2 expression bf X/xRy.
- **X** may be less than zero if **y** is an approximation of a rational number with an odd denominator.

APL Usage

```
apl> 2 * -3 -2 -1 0 1 2 3        apl> 64*%I6

0.125 0.25 0.5 1 2 4 8           64 8 4.0 2.8284 2.2974 2

apl> 10 * 0 1                    apl> 5 * -2

1 10                              .04

apl> -16 * % 2                   apl> -125 * % 3

0j4                              -5
```

```
apl> 16 * % 2                     apl> 125 * % 3

4                                 5

apl> 0j2 * 3

0j-8
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dyadic; Endoper; Litvect; Power; Quadout;
     #include "aplinit.h"

     endoper(quadout(dyadic(power,
          litvect("2 2 2 10"),
          litvect("0 1 2 2"))));
}

1 2 4 100
```

See Also

Comparison Tolerance (Lct), Exponential (*).

PRINTING PRECISION ⎕PP **Lpp**

Lpp specifies the default **Printing Precision**, or number of significant digits, when displaying numbers. IBM's APL2 will accept a value for ⎕pp up to the maximum precision allowed by the system. CAPLIB2 takes a more conservative approach, hopefully to provide more consistent results across platforms. Anything beyond a value of 10 when assigned to Lpp is ignored.

APL Usage

```
apl> Lpp#10
apl> a#456789 456789.1 .0000005678
apl> a
```

```
456789 456789.1 0.00000057

apl> Lpp#4
apl> a

456789 456789       5.678e-7

apl> Lpp#10
apl> 2 2R1j2 3j4 5j6 7j8

1j2 3j4
5j6 7j8

apl> Lpp#10
apl> 2 2R0j1 2j3 4j5 6j0

0j1 2j3
4j5 6

apl> L#a#2 2R2j0 2j3 2j3e45 3e45j2

2                2e0j3
2j3e45           3e45j2

apl> Lpp#5
apl> a

2                2e0j3
0j3e45           3e45
```

PROMPT REPLACEMENT ⎕PR **Lpr**

Quad pr is an APL2 system variable not implemented in CAPLIB2.

PRINTING WIDTH ⎕PW **Lpw**

Quad pw is an APL2 system variable not implemented in CAPLIB2.

QUAD INPUT □ L

Quadin obtains an expression from the standard input file, evaluates it and returns the result. Empty input, one containing nothing other than tabs, newlines, and spaces, is ignored, and a new prompt (L:) is issued. System commands entered will be executed, then the prompt will be re-issued.

APL Usage

```
apl> z#4XL

L: 3+2

apl> z

20
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Quadin; Quadout;
     #include "aplinit.h"

     printf("Key an expression");
     endoper(quadout(quadin()));
}

Key an expression

L:+/I5

15
```

See Also

Assign (#), Index Generator (I), Plus (+), QQUADIN, Times (X).

QUAD OUTPUT □ L

Quadout prints a formatted copy of the APL variable argument, then returns the argument to its caller. **Quadout** is one of the few CAPLIB2 routines that doesn't free memory for an **aplcb** passed as argument that is marked temporary (see also discussion of subroutines **perm** and **temp**.

APL Usage

```
apl> 1 + L # 2 * 1 + L # 3 _ 1

2

8

9
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Quadout;
     #include "aplinit.h"

     endoper(quadout(litvect("1.23e+3 23.4 5")));
}

1230 23.4 5
```

See Also

Assign (#), Minus (_), Plus (+), QQUADOUT.

RIGHT ARGUMENT □R Lr

Quad r is an APL2 system variable not implemented in CAPLIB2. In APL2, it yields the array value of the right argment of the first line of the state indicator.

QUOTE QUAD INPUT ⍞ Q

Accepts character input from standard input, returning an APL character vector with the result.

APL Usage

Following is a transcription of a session using the APL interpreter. On the first line the command "`apl`" is keyed to invoke the interpreter, which prompts for input with the string "`APL>`". The statement "`x#Q`" invokes **quote quad (Q)** to get input, then assigns that input to variable **x**. Entering **x** at the second "`APL>`" prompt tells the interpreter to display the value for variable **x**.

```
C:>apl

APL> x#Q

how now brown cow

APL> x

how now brown cow
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Qquadin; Quadout;
     #include "aplinit.h"

     endoper(quadout(qquadin()));
}

I entered this, which will be echoed back.

I entered this, which will be echoed back.
```

See Also

Assign (#), Quad Output (L).

QUOTE QUAD OUTPUT ⍞ Q

Qquadout offers a means to display on standard output without a concluding *new line* signal ('\n' in C).

APL Usage

Following is a transcription of a session using the APL interpreter. Annotations are in italics, surrounded in parentheses.

```
C:>apl
```

(Invoke the APL interpreter from the DOS prompt.)

```
APL> Lcr Lfx 3 13 R 'z#prompt msg Q#msg        z#Q           '
```

(The APL interpreter prompts with the string "`APL>`*". The input defines a user function to prompt for input without defining a new line, then displays the canonical representation of the user function. The APL interpreter responds by displaying the three lines below.)*

```
   z#prompt msg
   Q#msg
   z#Q
APL> r # prompt 'Yes Master? '
```

(Invokes the user function, and assigns the result to the variable named "r".)

```
Yes Master? hi there
```

(In response to the prompt, "hi there" was keyed.)

```
APL> r
```

("r" is keyed to display it's content.)

```
    hi there
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Execute; Qquadout; Qquadin; Quadout;
```

```
     #include "aplinit.h"

     endoper(qquadout(chrvect("key an expression ")));
     endoper(execute(qquadin()));
}
```

```
key an expression 2*3

8
```

See Also

Assign (#), Canonical Representation (Lcr), Function Establishment (Lfx), Quad Output (L), Reshape (R).

RANDOM LINK ⎕RL **Lrl**

In CAPLIB2, assigning a value to **Lrl** starts a repeatable sequence of pseudo-random numbers in the **roll** and **deal** functions. In IBM's APL2, the value in Lrl changes each time you use either the roll or deal function, where in CAPLIB2 its value remains constant.

APL Usage

```
apl> Lrl#23
apl> ?10R10

9 7 4 5 5 6 4 10 3 3

apl> Lrl

23

apl> 10?10

3 8 2 10 1 4 9 6 5 7

apl> 10?10

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

```
apl> Lrl#23 " Assignment of 23 starts the same sequence
apl> ?10R10

9 7 4 5 5 6 4 10 3 3

apl> 10?10

3 8 2 10 1 4 9 6 5 7

apl> Lrl

23
```

See Also

Deal (?), Roll (?).

RANK

Rank is the number of dimensions an APL variable has. The APL expression **RRx** gives the rank of variable **x**. Scalars have rank 0, signifying they have no structure, vectors have rank 1, meaning data is arranged along a single dimension, matrices rank 2, with data arranged in rows and columns, and so forth.

See Also

Reshape (R), Shape (R).

RAVEL , **,**

Monadic **ravel**, denoted by a **comma**, produces a vector whose elements are from the argument in row major order, if no axis is specified.

Ravel APL Usage

```
apl> , L # 3 4 R I 12

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

```
apl> L#a#3 3RI9

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

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

apl> ,L#3 4 R ’abcdefghijkl’

abcd
efgh
ijkl

abcdefghijkl

apl> L#b#2 2 4R’bad fog go  slow’

bad
fog

go
slow

apl> Rb

2 2 4

apl> L#m#,b

bad fog go  slow

apl> Rm

16
```

```
apl> L#z#,a

1 2 3 4 5 6 7 8 9

apl> Rz

9

apl> Rc#4

.

apl> ‘=c

0

apl> Rw#,c

1

apl> ‘=w

1
```

Ravel Selective Specification

Ravel can be used for selective specification, also described on page 48:

```
apl> L#s#2 2R(1 2) (3 4) (5 6) (7 8)

 1 2  3 4
 5 6  7 8

apl> (,s)#’abcd’
apl> s
```

```
apl> ‘=s

2

apl> Rs
```

```
  abcd abcd    abcd abcd         2 2
  abcd abcd    abcd abcd

apl> ‘=s

3
```

Ravel C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Litvect;
     Quadout; Ravel; Reshape;
     #include "aplinit.h"

     endoper(quadout(ravel(quadout(reshape(
       litvect("3 4"),
       chrvect("abcdefghijkl"))))));
}

abcd
efgh
ijkl

abcdefghijkl
```

Ravel with Axis

As near as I can tell, ravel with axis is just a shorthand method of reshape. This CAPLIB2 expression always produces 1:

```
((R,[x]r)Rr)‘=,[x]r
```

Ravel with axis falls into three categories:

1. If **x** is a fraction, a new axis of length 1 is formed;
2. If **x** is an integer, the dimensions or **r** indexed by **x** are combined;
3. When **x** is empty a new last axis of length 1 is created.

Ravel with Axis, APL Usage

```
apl> L#a#2 3R'tensix'

ten
six

apl> L#y#,[.1]a

ten
six

apl> Ry

1 2 3

apl> L#w#,[2.1]a

t
e
n

s
i
x

apl> L#v#,[1.1]b

10
15
20

apl> L#p#,[2 3]c

 1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24

apl> Rp

3 8
```

```
apl> L#z#,a

tensix

apl> Rz

6

apl> L#x#,[1.1]a

ten

six

apl> Rx

2 1 3

apl> Rw

2 3 1

apl> b#10 15 20

apl> Rv

3 1

apl> L#c#3 2 4RI24

 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24
```

```
apl> L#j#,[1 2]c

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

```
apl> Rj

6 4
```

```
apl> a#'ant' 'boar' 'cat' 'dog' 'elk' 'fox' 'gnu'
apl> b#'hen' 'ibex' 'jird' 'kite' 'lamb' 'mice'
apl> c#'nene' 'ox' 'pig' 'quail' 'rat' 'seal'
apl> L#d#4 2 3Ra,b,c,'titi' 'viper' 'wolf' 'yak' 'zebra'

 ant  boar  cat
 dog  elk   fox

 gnu  hen   ibex
 jird kite  lamb

 mice nene  ox
 pig  quail rat

 seal titi  viper
 wolf yak   zebra

apl> L#h#2 3RI6

1 2 3
4 5 6

apl> L#n#,[I0]h

1
2
3

4
5
6

apl> k#'prune' 'pear' 'fig'
```

```
apl> L#m#,[1 2]d

 ant  boar  cat
 dog  elk   fox
 gnu  hen   ibex
 jird kite  lamb
 mice nene  ox
 pig  quail rat
 seal titi  viper
 wolf yak   zebra

apl> Rm

8 3

apl> `=m

2

apl> Rn

2 3 1

apl> `=n

1
```

```
apl> L#i#,[IO]k

 prune
 pear
 fig

apl> e#3 2 5RI30
apl> ,[IRRe],[.5]e

 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
```

```
apl> Ri

3 1

apl> ‘=i

2

apl> g#’jim’ ’ed’ ’mike’
apl> L#f#,[IRRg],[.5]g

 jim ed mike

apl> Rf

1 3
```

Ravel with Axis and Selective Specification

Ravel with axis can be used for selective specification, also described on page 48:

```
apl> q#2 3 4RI24
apl> (,[2 3]q)#2 12R_I24
apl> q

 -1  -2  -3  -4
 -5  -6  -7  -8
 -9 -10 -11 -12

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

```
apl> Rq

2 3 4
```

C Usage, Ravel with Axis

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Litvect;
     Quadout; Ravel2; Reshape; Shape;
     #include "aplinit.h"
```

```
     endoper(quadout(shape(quadout(ravel2(
          reshape(litvect("2 3"),chrvect("tensix")),
          litvect(".1")
          )))));
}
ten
six
1 2 3
```

See Also

Assign (#), Enlist (E), Index Generator (I), Quad Output (L), Reshape (R).

)READ

)read file

The named DOS or Linux file is read. Each line delimited by a newline character (*nl* in C), becomes a row of the character variable named *file.*

)READ isn't standard APL usage. I added it because I don't like the line editor offered by standard APL, preferring to use my own. I'll edit perhaps the canonical representation of a function in a DOS or Linux text file, then invoke APL, use the)READ command, then enter " Lfx file" to fix the function for later execution.

See Also

)WRITE

REAL

C programming usage only, **real** returns a copy of the APL variable received as argument, converting to double floating point if the input is integer, and indicating an error if it is character.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
```

```
main()
{
     Endoper; Litvect; Quadout; Real;
     #include "aplinit.h"

     endoper(quadout(real(quadout(
       litvect("1.25 7.89 -3.45")))));
}

1.25 7.89 -3.45

1.25 7.89 -3.45
```

RECIPROCAL ÷ %

Reciprocal returns its argument divided into 1.

APL Usage

```
apl> %4 1 .2 -3                    apl> %2j2 0j1 0j-1

 .25 1 5 -0.33333333               .25j-0.25  0j-1 0j1
```

C Usage

Subroutine **recip** accepts an argument of type **double**, and returns a value of type **double**.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Monadic; Quadout; Recip;
     #include "aplinit.h"

     endoper(quadout(monadic(recip,
          litvect("4 -5"))));
}

0.25 -0.2
```

REDUCE ∤ / ‘/ /

Given **v** is a vector of length **n**, the APL expression **+/v** is equivalent to -

$$\sum_{i=1}^{n} v_i$$

If **v** were set to 1 2 3, the result of **+/v** would be 6. **Reduction** is known as an **operator** since it can be applied to any dyadic function (in classic APL, only scalar dyadics were allowed) to derive a different function. Examples are the expression **X/1 2 3**, which produces 7, and **S/1 2 3**, producing 3.

Reduction of an empty vector is defined to be the **identity** element as described on page 177 when the function given to reduce as argument is scalar dyadic. For nonscalar functions, IBM's APL2 uses the value returned by passing the prototype of the right argument to reduce as argument to a related **identity function**. This is not implemented in CAPLIB2. Passing an empty array to reduce along with a nonscalar dyadic function produces an error:

```
apl> R/2 0R1

R/2 0R1
 ^ 999 - internal error
```

Reduction of a scalar or one-element vector is the single item itself. Reduction of larger vectors is done by evaluating an expression derived by placing the dyadic function given as argument between items of the vector. As in all APL expressions, the order of execution is from right to left. You might expect _ **/1 2 3** to produce −4. Instead, it produces 2.

Variables of rank 2 or more are treated as sets of vectors aligned on an axis, with reduction taking place along each vector. In the following CAPLIB2 expression, replacing **r** with any appropriate matrix, **i** with any appropriate index values, and **X** with any appropriate dyadic function, will always produce 1 for an answer:

```
apl> (X/r[i;]) ‘= (X/r#2 3RI6)[i#2]

1
```

the same is true for similar expressions for 3-dimensional arrays, and so forth:

```
apl> (X/r[i;j;]) ‘= (X/r#2 3 4RI24)[i#2;j#1 2]

1

apl> (+/r) ‘= ‘X+/‘1‘Z[RRr]r#2 3 4 5RI120

1
```

You can specify the axis of reduction. If you don't, the default is the **last** dimension for *slash (/)*, and the **first** for *slash bar (‘/)*. The form is **FR**[**a**]**v**, where F is the dyadic function (+, ∗, etc.), R is either of the symbols " / " or "‘/", and the bracketed value **a** is the axis index. Depending on the **index origin**, the axis index is relative 0 or 1 (see the explanation on page 184).

The shape of the result is the shape of the argument with the axis of reduction removed. If the dimension is 1 along the axis of reduction, the result is identical in value, the only difference being the shape. If the argument is nested, the depth of the result is the same.

APL Usage

```
apl> L # m # 3 4 R I 12

1  2  3  4
5  6  7  8
9 10 11 12

apl> + ‘/ [2] m

    10 26 42

apl> + / [2] m

10 26 42

apl> L#z#+/(1 2)(3 4)(5 6)

 9 12

apl> Rz

.
```

```
apl> + / m

10 26 42

apl> + ‘/ m

15 18 21 24

apl> + / [2] m

15 18 21 24

apl> +/1 2 3 4 5

15

apl> V/0 0 1 1 0

1

apl> L#w#,/’ab’ ’cd’ ’ef’
```

```
apl> ‘=z

2

apl> L#m#3 4RI12

1  2  3  4
5  6  7  8
9 10 11 12

apl> +/m

10 26 42

apl> L#r#3 2R’acegik’,‘1’bdfhfl’

 ab cd
 ef gh
 if kl

apl> L#y#,/r

 abcd efgh ifkl

apl> Ry

3

apl> ‘=y

2

apl> L#a#X/2 3 0R‘Z0 0

 0 0  0 0  0 0
 0 0  0 0  0 0

apl> Ra

2 3
```

```
 abcdef

apl> Rw

.

apl> ‘=w

2

apl> =/15

15

apl> L#n#4 1R2 4 6 8

2
4
6
8

apl> %/n

2 4 6 8

apl> +/I0

0

apl> L#m#3 4RI12

1  2  3  4
5  6  7  8
9 10 11 12

apl> +/[1]m

15 18 21 24

apl> X/[1]m
```

```
apl> `=a

2

apl> L#n#2 3 4RI24

 1  2  3  4
 5  6  7  8
 9 10 11 12

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

apl> ,/[1]2 3RI6

 1 4  2 5  3 6

apl> L#n#2 1 4R2XI8

 2  4  6  8

10 12 14 16

apl> %/[2]n

 2  4  6  8
10 12 14 16
```

```
45 120 231 384

apl> X`/m

45 120 231 384

apl> +/[1]n

14 16 18 20
22 24 26 28
30 32 34 36

apl> +/[2]n

15 18 21 24
51 54 57 60

apl> %/[2]2 0 3R0

1 1 1
1 1 1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Plus; Quadout; Reduce;
     double identity=0.0;
     int axis=1;
     #include "aplinit.h"

     endoper(quadout(reduce(plus,&identity,
       litvect("1.25 7.89 -3.45"),axis)));
```

```
}
```

5.69

Here's a second example that uses CAPLIB^2 routine **reducef** to handle reduce independent of data type:

```
#define INCLUDES APLCB+APLCHDEF+APLMAIN+STDIO+APLDERIV
#include "includes.h"
main()
{
     Dervfree; Endoper; Indxsub; Litvect; Newderiv;
     Reducef; Reshape; Quadout;
     void subfill(int, Deriv_sub*);
     Aplderiv dp;
     #include "aplinit.h"

     dp = newderiv(NULL,NULL,NULL); /* Allocate. */
     dp->deriv_flags |= DERPERM; /* Mark it permanent. */
     dp->deriv_func = reducef; /* Reduce all data types. */
     subfill(SLASH, &(dp->deriv_left));
     dp->deriv_op = dp->deriv_left.fun;
     subfill(PLUS,  &(dp->deriv_left));
     endoper(quadout(dp->deriv_func(dp,NULL,
          reshape(litvect("3 2"),indxsub(6)))));
     endoper(quadout(dp->deriv_func(dp,NULL,
          litvect("1j2 3j4 5j6"))));
     dp->deriv_flags &= ~DERPERM;
     dervfree(dp);
}

void subfill(code,sub)
int code;
Deriv_sub *sub;
{
     Funexec; Pickdyad;
     struct apltoken tok;
     Apltoken tokptr;

     tok.token_code = code;
     tok.token_flags=0; /* So funexec won't copy token. */
     tokptr = funexec(&tok); /* Get info. on code. */
     sub->funcode = tok.token_code;
```

```
     sub->fun = tok.token_ptr.token_function;
     sub->sdp = pickdyad(sub->fun);
}
```

```
 3 7 11

 9j12
```

See Also

Aplderiv.h, Assign (#), Disclose(‘X), Each(‘1), Enclose(‘Z), Ireduce, N-wise (/ , ‘/), Plus (+), Quad Output (L), Scan (\ ).

REDUCE N-WISE ⌿ / ‘/ /

CAPLIB2 only. This is reduce with a left argument that dictates how many items along the axis of reduction are to participate to produce a given item of the result. The format is -

```
z#n F/[x] r
```

The bracketed axis is optional. If specified, it must be either a simple scalar or one-item integer vector. If you don't supply it, the default, like **reduce**, is the **last** dimension for *slash (/)*, and the **first** for *slash bar (‘/)*.

The left argument **n** must be either a scalar or one-item integer vector. Let **q** be the magnitude or absolute value of **n**. At each item of the right argument **r**, starting at the left end of the axis dimension, reduction happens for **q** items of **r** to produce one item of the result, until there are no longer **q** items remaining along the axis of reduction in **r**. If **n** is negative, the order of each **q** items selected is reversed before reduce is applied to produce a result.

APL Usage

```
apl> +/1 2 3 4

10

apl> 2=/2 3 4R'a'

1 1 1
```

```
apl> 1+/1 2 3 4

1 2 3 4

apl> 2+/1 2 3 4

3 5 7
```

```
1 1 1
1 1 1

1 1 1
1 1 1
1 1 1

apl> 3=/2 3 4R'a'

0 0
0 0
0 0

0 0
0 0
0 0

apl> 2,/1 2 3 4

 1 2  2 3  3 4

apl> r#1 2 3 4 5 6
apl> 6+/r

21

apl> 5+/r

15 20

apl> 4+/r

10 14 18

apl> 3+/r

6 9 12 15

apl> 2+/r

3 5 7 9 11

apl> 2^=/2 3 4R'a'

0 0 0
0 0 0
0 0 0

0 0 0
0 0 0
0 0 0

apl> 3^=/2 3 4R'a'

1 1
1 1
1 1

1 1
1 1
1 1

apl> 2+/(1 2)(3 4)(5 6)

 4 6  8 10

apl> 2+/3 4RI12

 3  5  7
11 13 15
19 21 23

apl> 2,/'abcdef'

 ab bc cd de ef

apl> 2,/3 3R'abcdefghi'

 ab bc
 de ef
 gh hi

apl>  3X/I6
```

```
apl> 1+/r

1 2 3 4 5 6

apl> -2_/1 4 9 16 25

3 5 7 9

apl> 2_/1 4 9 16 25

-3 -5 -7 -9

apl> -2,/'abcdef'

 ba cb dc ed fe

apl>  2,/'abcdef'

 ab bc cd de ef


apl> L#c#3 2R(1 2)(3 4)(5 6)(7 8)(9 10)(11 12)

  1 2    3 4
  5 6    7 8
 9 10  11 12

apl> 2X/[1]c

  5 12  21 32
 45 60  77 96

apl> -2_/10 20 30 40

10 10 10

apl> -2_/10 8 20 -3

-2 12 -23
```

```
6 24 60 120

apl> -3X/I6

6 24 60 120

apl> 0X/I5

1 1 1 1 1

apl> r#4 3RI12
apl> 4+/[1]r

22 26 30

apl> 3+/[1]r

12 15 18
21 24 27

apl> 2+/[1]r

 5  7  9
11 13 15
17 19 21

apl> 1+/[1]r

 1  2  3
 4  5  6
 7  8  9
10 11 12

apl> 0+/[1]r

0 0 0
0 0 0
0 0 0
0 0 0
```

```
0 0 0
```

C Usage

```
#define INCLUDES APLCB+APLCHDEF+APLMAIN+STDIO+APLDERIV
#include "includes.h"
main()
{
     Chrvect; Dervfree; Endoper; Iscalar; Newderiv;
     Reducef; Quadout;
     void subfill(int, Deriv_sub*);
     Aplderiv dp;
     #include "aplinit.h"

     dp = newderiv(NULL,NULL,NULL); /* New aplderiv. */
     dp->deriv_flags |= DERPERM; /* Mark it permanent. */
     dp->deriv_func = reducef; /* Reduces all data types. */
     subfill(SLASH, &(dp->deriv_left));
     dp->deriv_op = dp->deriv_left.fun;
     subfill(COMMA, &(dp->deriv_left));
     endoper(quadout(dp->deriv_func(dp,
          iscalar(-2),
          chrvect("abcdef"))));
     endoper(quadout(dp->deriv_func(dp,
          iscalar(2),
          chrvect("abcdef"))));
     dp->deriv_flags &= ~DERPERM;
     dervfree(dp);
}

void subfill(code,sub)
int code;
Deriv_sub *sub;
{
     Funexec; Pickdyad;
     struct apltoken tok;
     Apltoken tokptr;

     tok.token_code = code;
     tok.token_flags=0; /* So funexec won't copy token. */
     tokptr = funexec(&tok); /* Get info. on code. */
     sub->funcode = tok.token_code;
```

```
     sub->fun = tok.token_ptr.token_function;
     sub->sdp = pickdyad(sub->fun);
}

 ba cb dc ed fe
 ab bc cd de ef
```

See Also

Reduce.

RELATIVE

Extends relative scalar dyadic functions such as **eq** and **ge**, where the arguments are double floating point, and the result is integer, to conformable pairs of APL variables item-by-item, in a fashion otherwise identical to the functioning of subroutine **dyadic** discussed on page 103.

APL Usage

```
a # 4 1 R 0 0 1 1
b # 4 1 R 0 1 0 1
c # (a & b), (a > b), a, (a < b), b, (a ^= b)
c # c, (a V b), (a ‘V b), (a = b), (1 ‘R b), (a Z b)
c # c, (2 ‘R a), (a K b), (a P b), (4 R 1)
e # 1, d # -1 Y R c # (F a, b), ' ', 3 0 F 0, c
c # (e R d Y 'a b                     a  F  b'), [1] c
c,[1](eRdY'        &  >     <    ^=  V ‘V  =     Z     K  P')

    a b                     a  F  b
    0 0   0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
    0 1   0  0  0  0  1  1  1  1  0  0  0  0  1  1  1  1
    1 0   0  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1
    1 1   0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1
             &  >     <     ^=  V ‘V  =     Z     K  P
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
```

```
     Endoper; Eq; Ge; Gt; Le; Litvect;
     Lt; Ne; Quadout; Relative;
     #include "aplinit.h"

     endoper(quadout(relative(eq, /* -----> 0 1 0 */
          litvect("2.0 3.0 4.0"),
          litvect("3.0 3.0 3.0"))));
     endoper(quadout(relative(ge, /* -----> 0 1 1 */
          litvect("2.0 3.0 4.0"),
          litvect("3.0 3.0 3.0"))));
     endoper(quadout(relative(gt, /* -----> 0 0 1 */
          litvect("2.0 3.0 4.0"),
          litvect("3.0 3.0 3.0"))));
     endoper(quadout(relative(le, /* -----> 1 1 0 */
          litvect("2.0 3.0 4.0"),
          litvect("3.0 3.0 3.0"))));
     endoper(quadout(relative(lt, /* -----> 1 0 0 */
          litvect("2.0 3.0 4.0"),
          litvect("3.0 3.0 3.0"))));
     endoper(quadout(relative(ne, /* -----> 1 0 1 */
          litvect("2.0 3.0 4.0"),
          litvect("3.0 3.0 3.0"))));
}
```

See Also

Assign (#), Axis ([]), Catenate (,), DYADIC.

RELATIONAL FUNCTIONS

The relational functions in APL are:

z # $l < r$	Less than
z # $l\,\mathrm{K}\,r$	Not greater
z # $l = r$	Equal
z # $l\,\mathrm{Z}\,r$	Not less
z # $l > r$	Greater
z # $l \wedge =r$	Not equal

For real numbers, the answers are pretty much what you'd expect. For any two real number scalars results are always either 1 or 0, indicating the truth or falseness of the relationship.

For character data, the only two functions permitted are **equal** and **not equal**.

For complex numbers, other than **equal** and **not equal**, the comparisons are done based on the magnitudes of the arguments.

Here are a few examples. For others, see the descriptions for each relational function elsewhere in this book:

```
apl> 'trial'='trial'

1 1 1 1 1

apl>8 -2 6 -4 0 < 0

0 1 0 1 0

apl> L#l#('in' 'out') (9 5 6) (`Z2 2RI4)

  in out   9 5 6   1 2
                   3 4


apl> L#r#('it' 'but') 6 (2 2R1 8 5 4)

  it but   6   1 8
               5 4


apl> )load ..\help\help

..\help\help.apl
^ loaded.
```

Entering e.g. help 'R' tells you what the R key does. Entering e.g. help 'reshape' tells you about the reshape function. The **display** function (see page 403) is also loaded with the **help.apl** workspace.

```
apl> display l=r

+$-----------------------------------------+
| +$--------------+ +$----+ +$------------+ |
| | +$--+ +$----+ | |0 0 1| U +$--+ +$--+ | |
| | |1 0| |0 1 1| | +~----+ | U1 0| U0 0| | |
```

```
| | +~--+ +~----+ |         | |0 0| |0 0| | |
| +E--------------+         | +~--+ +~--+ | |
|                           | +$--+ +$--+ | |
|                           | U0 0| U0 0| | |
|                           | |0 0| |0 1| | |
|                           | +~--+ +~--+ | |
|                           +E------------+ |
+E------------------------------------------+
```

REPLICATE ≠ \ ‘/ /

CAPLIB2 and APL2 only, did not exist in classic APL. The format is identical to **compress**, and if **lop** is restricted to 1s and 0s, its function is the same:

```
z#lop/[x]r
```

for which these statements are true:

- The bracketed axis specification is optional. The default axis is the last dimension of **r**.
- The rank of **z** is the same as **r**. All dimensions of **z** except the last match the corresponding dimensions of **r**. The depth of **z** and **r** match, unless a zero in **lop** should eliminate a nested item (see below).
- **lop** must be either a simple scalar or vector of integers, according to these rules:
  - If an item of **lop** is positive, the corresponding subarray of **r** is replicated that many times.
  - If an item of **lop** is negative, the fill item of the corresponding subarray of **r** is replicated according to the absolute value of the item of **lop**. The fill item is determined by the **type** (see page 15) of the first item of the subarray.
  - If an item of **lop** is zero, the corresponding subarray of **r** is dropped.
- **lop** and **r** are conformable if the axis dimension is equal to the sum of the count of nonnegative values in **lop**, i.e. $+/lopZ0$. But if **lop** or **r** is scalar, or if the axis dimension of **r** is 1, these extensions are first applied:

- If **lop** is a scalar or one-item vector, it is extended to $-1Y1, Rr$.
- If **r** is a scalar, it is treated as a one-item vector.
- if the axis dimension of **r** is 1, it is replicated along that axis $+/lopZ0$ times.

APL Usage

```
apl> 1 2 3 4 / 'abcd'

abbcccdddd

apl> 2 -1 1 -2 / r

 a a   8
 7 7 0 6 0 0
 5 5 0 4 0 0

apl> 0 2 0 1 / 'soap'

oop

apl> L#s#,[I0]'ton'

t
o
n

apl> L#w#'i' 'id' ('ide' 'idea')

 i id   ide idea

apl> '=w

3

apl> L#x#3 2 1 / w

 iii id id   ide idea

apl> '=x
```

```
apl> L#r#3 2R'a' 8 7 6 5 4

 a 8
 7 6
 5 4

apl> 1 2 -1 3 -2 / 6 7 8

6 7 7 0 8 8 8 0 0

apl> 2 / 4 5

4 4 5 5

apl> 1 -2 2 / s

t  tt
o  oo
n  nn

apl> 1 -2 3 / 6

6 0 0 6 6 6

apl> L#r#3 2 4RI24

 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24
```

```
3

apl> L#p#1 2 0 / w

 i id id

apl> ‘=p

2

apl> L#s#3 2 4 R I24

 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24

apl> 2/[2]s

 1  2  3  4
 1  2  3  4
 5  6  7  8
 5  6  7  8

 9 10 11 12
 9 10 11 12
13 14 15 16
13 14 15 16

17 18 19 20
17 18 19 20
21 22 23 24
21 22 23 24

apl> L#m#3 4RI12

1  2  3  4
5  6  7  8
```

```
apl> 2 -1 1 / [2]r

 1  2  3  4
 1  2  3  4
 0  0  0  0
 5  6  7  8

 9 10 11 12
 9 10 11 12
 0  0  0  0
13 14 15 16

17 18 19 20
17 18 19 20
 0  0  0  0
21 22 23 24

apl> t#’abcdefghijkl’
apl> L#t#3 1 4Rt

abcd

efgh

ijkl

apl> Rt

3 1 4

apl> -1 1/[2]t


abcd


efgh


ijkl
```

```
9 10 11 12

apl> 1 0 2 -1 ‘/ m

1  2  3  4
9 10 11 12
9 10 11 12
0  0  0  0
```

```
apl> R-1 1/[2]t

3 2 4

apl> 1 0 2 -1 /[1] m

1  2  3  4
9 10 11 12
9 10 11 12
0  0  0  0
```

```
apl> L#d#2 2 2R'he' 'me' 'we' 'us' 'i' 'a' 'o' 'e'

 he me
 we us

 i  a
 o  e

apl> Rd

2 2 2

apl> ‘=d

2
```

```
apl> L#j#0 2/[1]d

 ia
 oe

 ia
 oe

apl> ‘=j

1
```

```
apl> L#w#2 -1 1 /[2] d

 he me
 he me

 we us

 i  a
 i  a

 o  e
```

```
apl> Rw

2 4 2

apl> ‘=w

2
```

C Usage

Note that the CAPLIB2 routine **compress** performs the replicate function as well.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Compress; Endoper; Indxsub; Quadout;
     #include "aplinit.h"

     endoper(quadout(compress(indxsub(4),
          chrvect("abcd"),1)));
}
```

See Also

Compress (/ , ‘/).

RESHAPE ρ R

Dyadic **reshape**, **a R b**, reshapes the right argument **b** to the shape specified by the left argument **a**. The left argument must have rank less than or equal to 1. If the number of elements in the result exceed those in the right argument, elements are repeated cyclically.

APL Usage

```
apl> 3 4 R I12

1  2  3  4
5  6  7  8
9 10 11 12

apl> L#z#2 3 4RI24

 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
```

```
apl> 3 4 R 'abcde'

abcd
eabc
deab

apl> 3 5RI24

 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15

apl> L#b#'up' 'on' 'to' 'by'
```

```
17 18 19 20
21 22 23 24

apl> Rz

2 3 4

apl> RRz

3

apl> L#x#3 8Rz

 1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24

apl> Rx

3 8

apl> RRx

2
apl> n#2 0 6R5 3 2 1
apl> Rn

2 0 6

apl> RRn

3
apl> L#h#(I0)Rv

5

apl> Rh

.

apl> RRh
```

```
 up on to by

apl> L#y#2 3Rb

 up on to
 by up on

apl> 5Rb

 up on to by up

apl> L#s#0 2RI0

.

apl> Rs

0 2

apl> L#v#5 6 7

5 6 7

apl> Rv

3

apl> RRv

1

apl> L#k#''Rv

5

Rk

.

apl> RRk
```

```
0                                0

apl> L#m#3 0R0                   apl> Rm

.                                3 0

apl> RRm

2
```

Reshape and Selective Specification

Reshape can be used for selective specification, also described on page 48:

```
apl> t#'growth'
apl> (2 3Rt)#2 3R_I6             apl> (4Rt)#'abcd'
apl> t                           apl> t

 -1 -2 -3 -4 -5 -6                 a   b   c   d  -5 -6

apl> (8Rt)#I8
apl> t

   7     8     3     4   5 6
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Litvect; Quadout; Reshape;
     #include "aplinit.h"

     endoper(quadout(reshape(
       litvect("3 4"),
       chrvect("abcdefghijkl"))));
}

abcd
efgh
ijkl
```

See Also

Index Generator (I).

RESIDUE | |

For real numbers, **Residue, a | b**, produces the remainder of dividing **a** into **b**, according to two rules:

- 1. If the left argument is zero, the result is the right argument.
- 2. Otherwise the result is the C language expression $right - left * floor(right/left)$.

For all numbers, including complex, the result is given by this CAPLIB2 expression:

```
b_aXDb%a+a=0
```

APL Usage

```
apl> -2 -1 0 1 2  |  2.385 -3.456 5.8 2.5 -4.3

-1.615 -0.456 5.8 0.5 1.7

apl> 10 | 8 10 -4 4j3           apl> 10 | 17

8   0   6   4j3                 7

apl> -10 7j10 .3 | 17 5 10      apl> 4j6 | 7j10

-3   -5j7 .1                    3j4
```

C Usage

The only difference between **imod** and **iresidu** is which parameter divides into which.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dyadic; Endoper; Idyadic; Imod; Iresidu;
```

```
     Litvect; Quadout; Residue; Residuep; Residuex;
     double left[2],rite[2],ret[2];
     #include "aplinit.h"

     endoper(quadout(dyadic(residue,
          litvect("0  1     5 5 -5"),
          litvect("-3 2.385 2 -2 2"))));
     endoper(quadout(idyadic(imod,
       litvect(" 5  5 -5  0  2  2  2 -2 -2"),
       litvect(" 2 -2  2 -3  8  3 -3  3 -3"))));
     endoper(quadout(idyadic(iresidu,
       litvect(" 2 -2  2 -3  8  3 -3  3 -3"),
       litvect(" 5  5 -5  0  2  2  2 -2 -2"))));
     *left = 2.1;
     *rite = 5.6;
     residuep(left,rite,ret);
     printf("residuep returned %f\n",*ret);
     *left = 0;
     *(left + 1) = 12.0;
     *rite = 4.0;
     *(rite + 1) = 0;
     residuex(left,rite,ret);
     printf("residuex returned %fj%f\n",*ret,*(ret+1));
}
```

```
-3 0.385 2 3 -3
1 -1 1 0 2 2 -1 1 -2
1 -1 1 0 2 2 -1 1 -2
residuep returned 1.400000
residuex returned -8.000000j0.000000
```

REVERSE ⌽ ⊖ ‘O‘R

The result of monadic **reverse** is equal to the argument except items appear in reverse order. A vector such as 1 2 3 will appear as 3 2 1. Arguments of rank greater than one are treated as sets of vectors along a particular axis. The expression **(‘O x)** denotes reversal with the axis along the **last** coordinate of **x**, and **(‘R x)** denotes reversal with the axis along the **first** coordinate of **x**. Both the expressions **(‘O[j]x)** and **(‘R[j]x)** denote reversal along the **j**th axis of **x**.

APL Usage

```
apl> L # a # 2 3 R I6

1 2 3
4 5 6

apl> ‘Oa

3 2 1
6 5 4

apl> ‘O ’abc’

cba
```

```
apl> ‘Ra

4 5 6
1 2 3

apl> ‘R[1]a

4 5 6
1 2 3

apl> ‘R[2]a

3 2 1
6 5 4
```

```
apl> ‘OL#b#(1 2) (3 4) (5 6)

 1 2  3 4  5 6

 5 6  3 4  1 2
```

```
apl> ‘OL#a#’desserts’

desserts

stressed
```

```
apl> ‘OL#c#3 5R’emit regaltide ’

emit
regal
tide

 time
lager
 edit

apl> L#a#2 3 1R’in’ ’out’ ’up’ ’right’ ’left’ ’down’

 in
 out
 up

 right
 left
 down
```

```
apl> ‘O[2]a                     apl> ‘O[1]a

 up                              right
 out                             left
 in                              down

 down                            in
 left                            out
 right                           up
```

Reverse and Selective Specification

Reverse can be used for selective specification, also described on page 48:

```
apl> L#d#3 4RI12               apl> (‘Od)#3 4R’stopspinoder’
                               apl> d
1  2  3  4
5  6  7  8                     pots
9 10 11 12                     nips
                               redo

apl> L#b#3 4RI12               apl> (‘O[1]b)#3 4R_I12
                               apl> b
1  2  3  4
5  6  7  8                     -9 -10 -11 -12
9 10 11 12                     -5  -6  -7  -8
                               -1  -2  -3  -4
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Indexof; Indexv; Litvect; Perm;
     Quadout; Reshape; Reverse; Temp;
     Aplcb a,j,m;
     #include "aplinit.h"

     a = perm(chrvect("abcdefghijklmnopqrstuvwxyz "));
     m = perm(chrvect("head chief"));
     j = quadout(perm(indexof(a,m)));
```

```
/*
                                8 5 1 4 27 3 8 9 5 6
*/
     endoper(quadout(indexv(a,reverse(j,indxorg),NULL)));
/*
                                fiehc daeh
*/
     endoper(temp(a));
     endoper(temp(j));
     endoper(temp(m));
}
```

See Also

Assign (#), Axis ([]), Quad Output (L).

RMONADIC

Extends scalar monadic functions such as **not** and **signum**, where the argument is double floating point, and the result is integer, to APL variables item-by-item, in a fashion otherwise identical to the functioning of subroutine **monadic** discussed on page 230.

APL Usage

```
apl> ~ 3 2 1 0

0 0 0 1
```

```
apl> X -3.1 0.0 4.5

-1 0 1
```

C Usage

```
}
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Not; Quadout; Rmonadic; Signum;
     #include "aplinit.h"

     endoper(quadout(rmonadic(not,
          litvect("0 1.1 2.2 3.3")))); /* -> 1 0 0 0 */
     endoper(quadout(rmonadic(signum,
```

```
          litvect("3.14 0 -3.14")))); /* --> 1 0 -1 */
}
```

See Also

MONADIC, Not (~), Times (X).

ROLL ? ?

In standard APL, the **roll** function returns a random selection from the whole numbers beginning with the **index origin** and ending with **index origin** plus **n** minus 1, **n** being the argument, which must be a positive integer. This version of **roll**, however, also accepts fractional real numbers, and returns a random selection of fractional real numbers.

CAPLIB2 implements the use of Lrl (random link, see page 266) slightly differently than does IBM's APL2, and Lrl was not honored at all in classic APL. Each time you assign the same value to **Lrl** the roll function will restart a repeatable sequence of pseudo-random numbers. But in IBM's APL, **Lrl** changes with each reference, where with CAPLIB2 it remains the same.

APL Usage

```
apl> ? 6 6 6                     apl> ? 6 6 6

4 2 2                            4 6 2

apl> ? 6 6 6

5 2 3

apl> ? 6.1 6.1 6.1

4.1346110416 2.0719260231 2.8826654866

apl> ? 6.1 6.1 6.1

4.2606433302 6.7805444502 2.0475386822

apl> Lrl#7                       apl> Lrl#9
apl> ?5R10                       apl> ?5R10
```

```
6 3 4 7 10                        7 4 4 5 7

apl> Lrl

9

apl> Lrl#16807                    apl> Lrl#16807
apl> Lio#0                        apl> Lio#1
apl> ?6R10                        apl> ?6R10

5 1 7 5 4 8                       6 2 8 6 5 9
```

C Usage

Subroutine **roll** accepts an argument of type **double**, and returns a value of type **double**.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Imonadic; Iroll; Irollp;
     Litvect; Monadic; Quadout; Roll; Rollp;
     double rite,ret;
     #include "aplinit.h"

     endoper(quadout(monadic(roll,
       litvect("23.0 23.0 23.0"))));
     endoper(quadout(imonadic(iroll,
       litvect("5 5 5 5 5"))));
     rite = 50.0;
     rollp(&rite,&ret);
     printf("rollp returned %f\n",ret);
     rite = 10;
     printf("roll returned %f\n",roll(rite));
}

12.819025 5.0416883 8.0985748
3 5 1 4 2
rollp returned 25.738304
roll returned 2.246986
```

See Also

Deal (?), Random Link (Lrl).

ROTATE ⌽ ⊖ ‘O‘R

The APL expression **(k‘Rx)** denotes rotation along the first axis of **x**, and **(k‘Ox)** along the last. Assuming the **index origin** is 1 and **x** is a vector, **(k‘Rx)** is equivalent to -

```
x[1+(Rx)|-1+k+IRx]
```

Both the expressions **(k‘O[j]x)** and **(k‘R[j]x)** denote rotation along the **j**th axis of **x**. The shape of **k** must equal the shape of **x** with the **j**th axis removed. Each vector along the **j**th axis of **x** is rotated according to the corresponding element of **k**.

APL Usage

```
apl> 2 ‘O 2 3 5 7 11

5 7 11 2 3

apl> L # m # 3 4 R I 12

1  2  3  4
5  6  7  8
9 10 11 12

apl> 1 2 3 ‘O [2] m

 2  3  4  1
 7  8  5  6
12  9 10 11

apl> 2‘OL#b#2 5R’angleaside’

angle
aside

glean
```

```
apl>  -2 ‘O 2 3 5 7 11

7 11 2 3 5

apl> 0 1 2 3 ‘O [1] m

1  6 11  4
5 10  3  8
9  2  7 12

apl> 1‘OL#a#1 2 3 4 5 6 7

1 2 3 4 5 6 7

2 3 4 5 6 7 1

apl> -2‘Oa

6 7 1 2 3 4 5

apl> -1‘OL#d#2 4R’acheinks’
```

```
ideas

apl> h#3 3R'ateeattea'
apl> -1 0 1‘OL#h

ate
eat
tea

eat
eat
eat

apl> 1 2‘OL#k#2 3R'cat' 'bear' 'pony' 'gnu' 'bird' 'fox'

 cat bear pony
 gnu bird fox

 bear pony cat
 fox  gnu  bird

apl> L#q#2 3R4 0 -1 -2 5 1

 4 0 -1
-2 5  1

apl> L#s#2 3 5R'taressmartearthsetonlagershear'

tares
smart
earth

seton
lager
shear

apl> Rs

2 3 5

apl> a#'betta' 'carp' 'eel' 'loach'
apl> b#'bay' 'cedar' 'elm' 'larch'
```

```
ache
inks

each
sink

apl> Rk

2 3

apl> ‘=k

2

apl> q ‘O s

stare
smart
heart

onset
lager
hears
```

```
apl> L#c#3 4 1Ra,b,'boa' 'cavy' 'eland' 'lion'

 betta
 carp
 eel
 loach

 bay
 cedar
 elm
 larch

 boa
 cavy
 eland
 lion

apl> L#u#3 1R'alfred' 'think' 'quick'

alfred
think
quick

apl> 1‘Ru

 think
 quick
 alfred

apl> -1‘Ru

 quick
 alfred
 think

apl> L#v#3 5R0 1 -1 2 -2 -3 -1 1 3 0 1 0 2 -1 3

 0  1 -1  2 -2
-3 -1  1  3  0
```

```
apl> 1‘O[1]c

 bay
 cedar
 elm
 larch

 boa
 cavy
 eland
 lion

 betta
 carp
 eel
 loach

apl> 1‘O[2]c

 carp
 eel
 loach
 betta

 cedar
 elm
 larch
 bay

 cavy
 eland
 lion
 boa

apl> w#'abcdefghijklmnopqrst'
apl> w#w,(I20)
```

```
 1  0  2 -1  3

apl> v‘O[2]w

  a  g  r  n  o
  k  b  m  d  j
  p  l  c  i  j
  p  b  m  i  j

  6 17  8 19  5
 11  7  8  4  5
 11 17  8  4  4
  1 12  3 14 20

  F  B  M  S  T
  F  L  C  S  T
  P  G  R  I  O
  A  Q  H  N  O
```

```
apl> x#'ABCDEFGHIJKLMNOPQRST'
apl> L#w#3 4 5Rw,x

  a  b  c  d  e
  f  g  h  i  j
  k  l  m  n  o
  p  q  r  s  t

  1  2  3  4  5
  6  7  8  9 10
 11 12 13 14 15
 16 17 18 19 20

  A  B  C  D  E
  F  G  H  I  J
  K  L  M  N  O
  P  Q  R  S  T

apl> Rw

3 4 5
```

Rotate and Selective Specification

Rotate can be used for selective specification, also described on page 48:

```
apl> 2‘OL#w#'stripe'

stripe

ripest
```

```
apl> (2‘Ow)#'thermo'
apl> w

mother
```

```
apl> y#3 4RI12
apl> (1 -1 2 -2‘O[1]y)#3 4R'ABCDEFGHIJKL'
apl> y

 IFGL
 AJKD
 EBCH
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
```

```
#include "includes.h"
main()
{
     Chrvect; Endoper; Iscalar; Litvect; Perm;
     Quadout; Reshape; Rotate; Temp;
     Aplcb m;
     #include "aplinit.h"

     endoper(quadout(  /* --------> 5 7 11 2 3 */
               rotate(
                    iscalar(2),
                    litvect("2 3 5 7 11"),
                    1 /* axis */
               )
          ));
     m=perm(quadout( /* -------------------> ABCD */
               reshape(                  /* EFGH */
                    litvect("3 4"),      /* IJKL */
                    chrvect("ABCDEFGHIJKL")
               )
          ));
     endoper(quadout( /* ------------------> AFKD */
               rotate(                   /* EJCH */
                    litvect("0 1 2 3"),  /* IBGL */
                    m,
                    1 /* axis */
               )
          ));
     endoper(quadout( /* ------------------> BCDA */
               rotate(                   /* GHEF */
                    litvect("1 2 3"),    /* LIJK */
                    m,
                    2 /* axis */
               )
          ));
     endoper(temp(m));
}
```

See Also

Assign (#), Index Generator (I), Indexing ([]), Plus (+), Residue (|), Reverse (‘O), Shape (R).

)RUN

)run file

The named DOS file is run. That is, each line delimited by a newline character (*nl* in C), is passed to the *execute* APL operator (see page 121) as a character vector.

)RUN isn't standard APL usage. I added it because I sometimes find the interactive environment offered by standard APL cumbersome. I often edit a series of APL statements in a DOS or Linux text file, then invoke APL and use the)RUN command to execute those statements.

See Also

)READ,)WRITE.

)SAVE

)save wsname

The active workspace in memory is written to the named DOS workspace file. Workspaces on disk under DOS have the qualifier *.apl*, which you should leave off on the)save command. You can however qualify the name with its directory path, e.g.)*save* *cap**handbook**wsname*.

You'll get an error message if you give a name on the)save command that matches an existing workspace name on disk, but doesn't match the name of the active workspace.

See Also

Appendix E.

SCALAR

Scalar is both the name of a CAPLIB2 routine and an attribute of APL variables.

C program usage only, the scalar CAPLIB2 routine creates an apl variable from a double floating point constant.

An Apl variable is a **scalar** if it consists of a single element and has no dimensions. For example, in the APL expression **a#23**, **a** becomes a scalar variable. Its rank, produced by the APL expression **RRa**, is zero.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Quadout; Scalar;
     #include "aplinit.h"

     endoper(quadout(scalar(2.34)));
}
```

```
2.34
```

See Also

Assign (#), ISCALAR, Shape (R).

SCAN +\ ‘\ \

Given **v** is a vector of length **n**, the APL expression + \ **v** produces a vector whose length is also **n**, and whose **k**th element is equal to the reduction over the first **k** elements of **v**. Reduction is explained on page 275. **Scan** is known as an **operator** since it can be applied to any scalar dyadic function to derive a different function.

Variables of rank 2 or more are treated as a set of vectors along an axis, in a manner identical to that for reduction. If you don't specify an axis, the default is the **last** dimension for *slope (\ )*, and the **first** for *slope bar (‘\ )*. The form for specifying the axis is the same as for reduction.

The shape of the result of scan is the shape of the argument.

APL Usage

```
apl> + \ 1 2 3 4 5

1 3 6 10 15

apl> _ \ 1 2 3 4 5
```

```
apl> X \ 1 2 3 4 5

1 2 6 24 120

apl> L # m # 3 4 R I 12
```

```
1 -1 2 -2 3

apl> +\m

1  3  6 10
5 11 18 26
9 19 30 42

apl> +\1 2 3 4 5

1 3 6 10 15

apl> V\0  0 1 1 0

0 0 1 1 1

apl> ,\2 3RI6

 1  1 2  1 2 3
 4  4 5  4 5 6

apl> ,\[1]2 3RI6

   1    2    3
 1 4  2 5  3 6

apl> +\[1]L#n#2 3 4RI24

 1  2  3  4
 5  6  7  8
 9 10 11 12

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

 1  2  3  4
 5  6  7  8
 9 10 11 12

14 16 18 20
```

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

apl> +\[1]m

 1  2  3  4
 6  8 10 12
15 18 21 24

apl> +\(1 2)(3 4)(5 6)

 1 2  4 6  9 12

apl> ,\'ab' 'cd' 'ef'

  ab abcd abcdef

apl> +\[1]L#m#3 4RI12

 1  2  3  4
 5  6  7  8
 9 10 11 12

 1  2  3  4
 6  8 10 12
15 18 21 24

apl> +\[2]n

 1  2  3  4
 6  8 10 12
15 18 21 24

13 14 15 16
30 32 34 36
51 54 57 60

apl> X\[1]m

 1   2   3   4
```

```
22 24 26 28                         5  12  21  32
30 32 34 36                        45 120 231 384

apl> X‘\m

 1   2   3   4
 5  12  21  32
45 120 231 384
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Plus; Quadout; Scan;
     double identity=0.0;
     int axis=1;
     #include "aplinit.h"

     endoper(quadout(scan(plus,&identity,
       litvect("1.25 7.89 -3.45"),axis)));
}

1.25 9.14 5.69
```

Here's a second example that uses CAPLIB2 routine **scanfrnt** to handle scan independent of data type:

```
#define INCLUDES APLCB+APLCHDEF+APLMAIN+STDIO+APLDERIV
#include "includes.h"
main()
{
     Dervfree; Endoper; Indxsub; Litvect; Newderiv;
     Reshape; Quadout; Scanfrnt;
     void subfill(int, Deriv_sub*);
     Aplderiv dp;
     #include "aplinit.h"

     dp = newderiv(NULL,NULL,NULL); /* Allocate. */
     dp->deriv_flags |= DERPERM; /* Mark it permanent. */
     dp->deriv_func = scanfrnt; /* Scan for all data. */
     subfill(SLOPE, &(dp->deriv_left));
```

```
     dp->deriv_op = dp->deriv_left.fun;
     subfill(PLUS,  &(dp->deriv_left));
     endoper(quadout(dp->deriv_func(dp,NULL,
          reshape(litvect("3 2"),indxsub(6)))));
     endoper(quadout(dp->deriv_func(dp,NULL,
          litvect("1j2 3j4 5j6"))));
     dp->deriv_flags &= ~DERPERM;
     dervfree(dp);
}

void subfill(code,sub)
int code;
Deriv_sub *sub;
{
     Funexec; Pickdyad;
     struct apltoken tok;
     Apltoken tokptr;

     tok.token_code = code;
     tok.token_flags=0; /* So funexec won't replicate token. */
     tokptr = funexec(&tok); /* Get info. on code. */
     sub->funcode = tok.token_code;
     sub->fun = tok.token_ptr.token_function;
     sub->sdp = pickdyad(sub->fun);
}
```

```
 1  3
 3  7
 5 11

 1j2 4j6 9j12
```

See Also

Assign (#), Axis ([]), Index Generator (I), ISCAN, Quad Output (L), Reduce (/), Reshape (R).

SETIX

Called to set an index level in the APL variable to be later passed to subroutine **indexm** as argument **indices**. **Setix** declares its arguments as follows:

```
struct aplcb *setix(left,level,index)
struct aplcb *left,*index;
int level;
```

Left points to an APL variable of type APLAPL that will later be passed to subroutine **indexm** as argument **indices**.

Level is an index to argument **left**. It identifies the dimension that argument **index** is for.

Index is a set of indices, or **NIL**.

See the programming example under the heading **INDEXING** on page 185.

See Also

INDEX ORIGIN, INDEXING, and INDEXM.

SHAPE ρ R

Monadic **shape**, **R b**, gives a vector containing the shape of the argument **b**. Applied to a scalar, **shape** yields an empty vector, equal to the CAPLIB2 expression **(I0)**.

APL Usage

```
apl> R 3 4 R 1

3 4

apl> L#a#1 2 3 'a' 'b' 4

 1 2 3 ab 4

apl> Ra

6

apl> RRa

1

apl> L#d#2 3 4R(I12),_I12
```

```
apl> R I 5

5

apl> L#b#'stand up'

stand up

apl> Rb

8

apl> L#c#3 4RI12

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

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

-1  -2  -3  -4
-5  -6  -7  -8
-9 -10 -11 -12

apl> Rd

2 3 4

apl> RRd

3
```

```
apl> L#q#('elsie' 'tom') 'hank' ('ed' 'bob' 'kim')

  elsie tom   hank   ed bob kim

apl> `=q

3

apl> Rq

3

apl> RRq

1

apl> R`1q

 2  4  3

apl> R`1`1q

  5  3              2  3  3

apl> R'a'
```

```
apl> Rc

3 4

apl> RRc

2

apl> L#h#'tom' 'ed' 'hank'

 tom ed hank

apl> `=h

2
```

```
apl> Rh

3

apl> R`1h

 3  2  4

apl> L#s#`Z 2 3RI6

 1 2 3
 4 5 6

apl> `=s

2

apl> Rs

.
```

```
.

apl> RR'a'

0

apl> R,'a'

1

apl> RR,'a'

1
```

```
apl> L#t#,‘Z2 3RI6

 1 2 3
 4 5 6

apl> ‘=t

2

apl> Rt

1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Litvect; Quadout; Shape;
     #include "aplinit.h"

     endoper(quadout(shape(litvect("3 4"))));
}

2
```

See Also

Index Generator (I), Reshape (R).

SHARED VARIABLES □SV **Lsv**

Share variables are not implemented in CAPLIB2. These variables, which also existed in classic APL, are used to communicate with functions outside the APL environment, and are supported in IBM's APL2 with a set of system variables:

- □SVC, Shared Variable Control Inquire and Set.

- $\square SVE$, Shared Variable Event.
- $\square SVO$, Shared Variable Offer Inquire and Set.
- $\square SVQ$, Shared Variable Query.
- $\square SVR$, Shared Variable Retraction.
- $\square SVS$, Shared Variable State.

)SI

Displays the state indicator, the stack of halted, i.e. suspended and pendent, functions. Their names are listed in order, the most recently halted first. Each is followed by the bracketed statement number that was about to be, or was being, executed when the function was halted.

Functions are halted in several ways: by an error, such as an attempt to divide by zero, or by the *stop control* discussed on page 159. They're called *suspended* when this happens, and are indicated in the state indicator list by *.

Pendent functions, those without * next to their names in the state indicator, are there because they've invoked another function that either got suspended, or became pendent by calling another function, and so forth until one finally did get suspended.

When a function gets suspended you get control at the terminal. You can enter pretty much any statement you like. You can clear functions on the state indicator from the last to the preceding suspension by entering a right arrow ($). Or you can branch anywhere in the topmost suspended function by entering a right arrow followed by the statement number to branch to.

You can resume execution at the point of suspension by branching to an empty vector, or to the line counter (Llc). This is a system global variable that is a vector holding the current statement numbers of the functions on the state indicator, in the same order, i.e., the most recently halted leftmost.

APL Usage

)si

C Usage

There is no C programming interface to)si.

See Also

Appendix E, Branching ($),)SINL, Stop control.

)SINL

)sinl

Displays the state indicator, the stack of halted, i.e. suspended and pendent, functions, along with a list of localized names. Other than the list of localized names **)sinl** is identical to **)si**.

See Also

Appendix E,)SI, Stop control.

SIGNUM × X

Signum is referred to as **Direction** in IBM's manual on APL2, which says it "Yields the number of magnitude 1 with the same phase as R for nonzero R," referring to the APL2 expression **"Z← × R."**

Similar in function to **signdbl** and **isign**, except **signum** accepts a double floating point argument, and returns an integer. It declares itself as follows:

```
int signum(x)
double x;
```

For any scalar real number **x**, the APL expression **Xx** yields -1 if $x < 0 - c$, 0 if $x = 0 \pm c$, and 1 if $x > 0 + c$, where c is the **comparison tolerance**, referred to in APL as the system variable **Lct**, and in a C program as **extern double fuzz**. By default, **fuzz** is initialized to the small value $1e - 13$.

APL Usage

```
apl> X -3.1 0.0 4.5
-1 0 1

apl> apl X3j4
```

```
apl> X-5
-1

apl> X0j1 0j-1
```

```
.6j.8                           0j1 0j-1

apl> X-4 0 4

-1 0 1
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Imonadic; Isign; Isignp; Litvect; Monadic;
     Rmonadic; Quadout; Signdbl; Signdblp; Signdblx;
     Signum;
     double rite[2],ret[2];
     int irite,iret;
     #include "aplinit.h"

     endoper(quadout(monadic(signdbl,
          litvect("3.14 0 -3.14"))));
     endoper(quadout(rmonadic(signum,
          litvect("3.14 0 -3.14"))));
     endoper(quadout(imonadic(isign,
          litvect("-3 0 4"))));
     irite = -4;
     isignp(&irite,&iret);
     printf("isign returned %d\n",iret);
     *rite = 50.0;
     signdblp(rite,ret);
     printf("signdblp returned %f\n",*ret);
     *(rite+1) = -5.0;
     signdblx(rite,ret);
     printf("signdblx returned %fj%f\n",*ret,*(ret+1));
     printf("signum returned %d\n",signum(-10.0));
}

1 0 -1
1 0 -1
-1 0 1
isign returned -1
```

```
signdblp returned 1.000000
signdblx returned 0.995037j-0.099504
signum returned -1
```

STATE INDICATOR

The state indicator is the stack of halted, i.e. suspended and pendent, functions. A complete discussion is found under the heading **)si**, which is the system command used to list the state indicator.

SYSTEM FUNCTIONS

The IBM implementation of APL2 contains 13 **system functions**, only 5 of which are implemented in CAPLIB2. They're all listed in the following table:

APL Symbol	CAPLIB2 Symbol	Exists in CAPLIB2?	Description
⎕AF	`Laf`	No	Atomic Function
⎕AT	`Lat`	No	Attributes
⎕CR	`Lcr`	Yes	Canonical Representation
⎕DL	`Ldl`	No	Delay
⎕EA	`Lea`	No	Execute Alternate
⎕EC	`Lec`	No	Execute Controlled
⎕ES	`Les`	No	Event Simulation
⎕EX	`Lex`	Yes	Expunge
⎕FX	`Lfx`	Yes	Fix
⎕NA	`Lna`	No	Name Association
⎕NC	`Lnc`	Yes	Name Class
⎕NL	`Lnl`	Yes	Name List
⎕TF	`Ltf`	No	Transfer Form

SYSTEM VARIABLES

The IBM implementation of APL2 contains 25 **system variables**, only 9 of which are implemented in CAPLIB2. They're all listed in the following table:

APL Symbol	CAPLIB2 Symbol	Exists in CAPLIB2?	Description
⎕	`L`	Yes	Evaluated Input/Output
⍞	`Q`	Yes	Character Input/Output
⎕AI	`Lai`	No	Account information
⎕AV	`Lav`	Yes	Atomic vector
⎕CT	`Lct`	Yes	Comparison tolerance (fuzz)
⎕EM	`Lem`	No	Event Message
⎕ET	`Let`	No	Event Type
⎕FC	`Lfc`	Yes	Format Control
⎕HT	`Lht`	No	Horizontal tab settings
⎕IO	`Lio`	Yes	Index Origin
⎕L	`Ll`	No	Left Argument
⎕LX	`Llx`	Yes	Latent expression, executed when workspace is loaded
⎕NLT	`Lnlt`	No	National Language Translation
⎕PP	`Lpp`	No	Printing precision
⎕PR	`Lpr`	No	Prompt Replacement
⎕PW	`Lpw`	No	Printing width
⎕R	`Lr`	No	Right Argument
⎕RL	`Lrl`	No	Random (link) number seed
⎕LC	`Llc`	Yes	Line counter: statement numbers of functions executing or halted
⎕TS	`Lts`	Yes	Time stamp
⎕TC	`Ltc`	No	Terminal Control
⎕TT	`Ltt`	No	Terminal Type
⎕TZ	`Ltz`	No	Time Zone
⎕UL	`Lul`	No	User Load
⎕WA	`Lwa`	No	Working area available

TAKE ↑ Y

The **Take** function `(s Y v)` can best be understood by first considering the simpler case where `s` is a non-negative scalar integer and `v` is a vector. The result is a vector consisting of the leftmost `(s D R v)` elements of `v`:

```
apl> 2 Y 1 2 3 4 5

1 2
```

If the left argument is negative, elements are taken from the end instead of the front:

```
apl> -2 Y 1 2 3 4 5

4 5
```

If **v** in (s Y v) is an array, **s** must be a vector with one element for each dimension of **v**. The **i**th element of **s** determines what is to be **take**n along the **i**th dimension of **v**:

```
apl> 2 3 Y 3 4 R 'abcdefghijkl'

abc
efg
```

The rank of the result equals the length of the left argument.

If an item of the left argument to take has a magnitude greater than the corresponding axis of the right argument, the extra positions in the result are set to a fill item (see page 15), which depends of the subarray.

Take with Axis

In APL2 and CAPLIB2, you can specify the axes along which **take** is to operate, with syntax like -

```
z # q Y [x] r
```

Take with axis is just like regular take, with values specified for the axes not specified such that the maximum is taken along them:

```
apl> m#2 3 4RI24
apl> a#1 3 2Ym
apl> b#1 2Y[1 3]m
apl> a‘=b

1
```

APL Usage

```
apl> L#y#4 5R'triadfieldmoosediner'

triad                           apl> -2 3Yy
field
```

```
moose
diner

apl> L#w#3 3 4R'beatmythantetonemeanhereuponweekdoes'

beat
myth
ante

tone
mean
here

upon
week
does

apl> 5Y21 33 52

21 33 52 0 0

apl> 5Y'red'

red

apl> L#u#2 3RI6

1 2 3
4 5 6

apl> L#n#(1 2) (3 4)

 1 2  3 4

apl> 4Yn

 1 2  3 4  0 0  0 0

apl> -6Y'a' 'a' 'b' 2

   aab 2
```

```
moo
din

apl> v#-1 -2 2
apl> L#z#vYw

we
do

apl> Rz

1 2 2

apl> -5Y21 33 52

0 0 21 33 52

apl> -5Y'red'

  red

apl> L#h#4 4Yu

1 2 3 0
4 5 6 0
0 0 0 0
0 0 0 0

apl> Rh

4 4

apl> -6Y1 'a' 2 'b'

 0 0 1 a 2 b
```

```
apl> 2 3Y0 2R‘Z0 0 0

  0 0 0   0 0 0   0 0 0
  0 0 0   0 0 0   0 0 0

apl> L#e#1Y2

2

apl> Re

1

apl> L#f#(I0)Y2

2

apl> Rf

.

apl> L#t#’t’ ’to’ (’tot’ ’tote’)

 t to   tot tote

apl> L#j#1Yt

 t

apl> ‘=j

1

apl> L#k#2Yt

 t to

apl> ‘=k

2

apl> L#a#3 5R’giantstoretrail’
```

```
apl> 3YI0

0 0 0

apl> L#g#1 1 1Y2

2

apl> Rg

1 1 1

apl> 2 3Y2

2 0 0
0 0 0

apl> L#s#8 ((6 5) (4 3))

 8   6 5  4 3

apl> ‘=s

3

apl> 3Ys

 8   6 5  4 3   0

apl> L#q#‘Os

  6 5  4 3   8

apl> 3Yq

  6 5  4 3   8   0 0  0 0

apl> L#b#2 3RI6

1 2 3
```

```
                                4 5 6
giant
store                           apl> 3Y[1]b
trail
                                1 2 3
apl> 2Y[1]a                     4 5 6
                                0 0 0
giant
store                           apl> L#h#2 3R'abcdef'

apl> 2 5Ya                      abc
                                def
giant
store                           apl> L#z#-4Y[1]h

apl> -3Y[2]a

ant                             abc
ore                             def
ail

apl> 3 -3Ya                     apl> Rz

ant                             4 3
ore
ail

apl> L#c#2 3R1'a' 3 4 5 6       apl> 4Y[1]c

 1 a 3                           1 a 3
 4 5 6                           4 5 6
                                 0 0 0
                                 0 0 0

apl> L#k#3 3 4R'herosheddimesodaboarpartlambtotodamp'

hero                            apl> -1 3Y[1 3]k
shed
dime                            lam
                                tot
soda                            dam
boar
```

```
part

lamb
toto
damp

apl> t#'d' 'do'('don' 'done')
apl> t#t,'m' 'me'('men' 'mene')
apl> L#s#2 3Rt

 d do   don done
 m me   men mene

apl> ‘=s

3

apl> L#h#2Y[2]s

 d do
 m me

apl> ‘=h

2
```

```
apl> -1 3Y[3 1]k

o
d
e

a
r
t

b
o
p

apl> L#j#1Y[1]s

 d do   don done

apl> ‘=j

3
```

```
apl> L#m#2 3R1(2 3)((4 5)(6 7))8(9 1)((2 3)(4 5))
```

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

apl> Rm

2 3

apl> L#t#1‘O[2]m

 2 3   4 5  6 7   1
 9 1   2 3  4 5   8

apl> 3Y[1]t
```

```
apl> 3Y[1]m

 1  2 3   4 5  6 7
 8  9 1   2 3  4 5
 0    0           0

apl> L#c#2 3 4 R 1 'a' 'b'

 1 a b 1
 a b 1 a
 b 1 a b

 1 a b 1
 a b 1 a
```

```
 2 3   4 5  6 7     1
 9 1   2 3  4 5     8
 0 0        0 0  0 0

apl> L#c#2 3 R 1 'a' 3 4 5 6

 1 a 3
 4 5 6

apl> L#d#4Y[1]c

 1 a 3
 4 5 6
 0 0 0
 0 0 0
```

```
 b 1 a b

apl> 4 5Y[2 3]c

 1 a b 1 0
 a b 1 a 0
 b 1 a b 0
 0 0 0 0 0

 1 a b 1 0
 a b 1 a 0
 b 1 a b 0
 0 0 0 0 0

apl> Rd

4 3
```

Take and Selective Specification

Take can be used for selective specification, also described on page 48:

```
apl> L#p#'abcde'

abcde

apl> (2Yp)#1 2
apl> p

 1 2 cde

apl> L#ky#3 4R'abcdefghijkl'

abcd
efgh
ijkl

apl> (-2 1Yky)#1 2
apl> ky

 a bcd
 1 fgh
```

```
 2 jkl

apl> L#u#3 4R'abcdefghijkl'

abcd
efgh
ijkl

apl> (-2Y[2]u)#3 2RI6
apl> u

 ab 1 2
 ef 3 4
 ij 5 6
```

C Usage

```
#define INCLUDES apl> CB+apl> MAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Iscalar; Quadout; Take;
     #include "apl> init.h"

     endoper(quadout(take(iscalar(2),chrvect("abcde"))));
}

ab
```

See Also

Drop (U), Reshape (R), Shape (R).

TEMP

C program usage only, marks an apl variable temporary. CAPLIB2 subroutine **temp** declares itself as follows:

```
struct aplcb *temp(rite)
struct aplcb *rite;
```

Aplcbs are used to represent APL variables. The complete description of an **aplcb** begins on page 29. Among the fields it contains is one called **aplflags**, which, among other things, can be set to designate the **aplcb** as being temporary. Subroutine **temp** ensures the temporary flag is turned on in the **aplcb** pointed to by its argument, then returns its argument to its caller.

Subroutines and functions described in this manual, such as DYADIC, MONADIC, QUADOUT, etc., that accept pointers to APLCBs as arguments, will call ENDOPER after they've completed their processing. Subroutines and functions that create APLCBs and return pointers to them, such as CHRVECT, CONVECT, and LITVECT, unless otherwise indicated in this manual, will mark them temporary. Thus, these APLCBs will be freed automatically the first time they are passed as an argument to DYADIC, MONADIC, etc., unless they are marked permanent by passing them to **perm**.

See also

APLCB, ENDOPER, GETCB, PERM.

TERMINAL CONTROL ⎕TC **Ltc**

IBM's APL2 has a system variable $\square TC$, a three-item vector giving the backspace, new line, and line feed characters of the current terminal in use. This is not implemented in CAPLIB2.

TIMES × **X**

Times is a scalar dyadic function that does double floating point multiplication.

APL Usage

```
apl> 2.2 X 3

6.6

apl> 3 X 0 -2 5 .7

0 -6 15 2.1
```

```
apl> 3 X 4

12

apl> 1j2 X 3j4

-5j10
```

```
apl> 1 -3 .8 X 1 .5 -.2

1 -1.5 -0.16
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Dyadic; Endoper; Idyadic; Iscalar;
     Itimes; Quadout; Scalar; Times;
     Timesp; Timesx;
     double left[2],rite[2],ret[2];
     #include "aplinit.h"

     endoper(quadout(dyadic(
       times,scalar(2.2),scalar(3.0))));
     endoper(quadout(idyadic(
       itimes,iscalar(2),iscalar(3))));
     *left = 2.1;
     *rite = 5.6;
     timesp(left,rite,ret);
     printf("timesp returned %f\n",*ret);
     *left = 2.0;
     *(left + 1) = 12.0;
     *rite = 4.0;
     *(rite + 1) = 1.5;
     timesx(left,rite,ret);
     printf("timesx returned %fj%f\n",*ret,*(ret+1));
}

6.6
6
timesp returned 11.760000
timesx returned -10.000000j51.000000
```

TIME STAMP ⎕TS **Lts**

Contains the current date and time. **Lts** is a simple integer vector with these seven items:

- Lts[1]: Current year, e.g. 1997
- Lts[2]: Current month, 1-12.
- Lts[3]: Current day of month, 1-31.
- Lts[4]: Current hour, 0-23.
- Lts[5]: Current minute, 1-60.
- Lts[6]: Current second, 1-60.
- Lts[7]: Current millisecond, 0-999.

APL Usage

```
apl> '0006/06/06 06:06:06'F6YLts

1996/12/07 21:48:08
```

See Also

Format (F), Time Zone (Ltz).

TIME ZONE ⎕TZ **Ltz**

Ltz indicates the offset in hours from Greenwich Mean Time (GMT), and the date and time reported by the system on which CAPLIB2 is being executed. Setting Ltz affects the hour reported by the time stamp system variable, **Lts**.

APL Usage

```
apl> Ltz

0

apl> '0006/06/06 06:06:06'F6YLts

1996/12/07 21:48:08

apl> Ltz#-1
```

```
apl> '0006/06/06 06:06:06'F6YLts

1996/12/07 20:48:09
```

See Also

Format (F), Time Stamp (Lts).

TRANSFER FORM ⎕TF Ltf

IBM's APL2 has a system variable $\square TF$, called "Transfer Form." This is not implemented in CAPLIB2. Use Lfx (Function Establishment) instead.

TRANSPOSE ⍉ T

Dyadic transpose, z#pTr, produces an array **z** from the elements of **r**, with the **i**th dimension of **r** being the **p**[**i**]th of **z**, so **(Rz)**[**p**] equals **Rr** if all possible indices of **r** were represented in **p**.

Argument **p** may consist only of the indices of the dimensions of **r**, but does not need to contain every possible index. If **p** contains index **n** it must also contain all positive integers less than **n**. For example, if **RRr** is 3, 1 1 2 and 1 1 1 are acceptable for **p**, but 1 3 1 isn't.

If **p** indexes all possible axes of **r**, the result **z** will contain all items from **r**, but with some permutation of the dimensions of **r**. Here are some examples illustrating how the axes mapping occurs:

```
apl> L#a#(aix#2 3 4)R'bearlynxduckponybirdoxen'

bear
lynx
duck

pony
bird
oxen

apl> Ra

2 3 4
```

```
apl> L#z#(zix#1 3 2)Ta

bld
eyu
anc
rxk

pbo
oix
nre
ydn
```

```
apl> L#w#(wix#2 1 3)Ta

bear
pony

lynx
bird

duck
oxen

apl> Rw

3 2 4

apl> tix#IRRa
apl> (ix#2 1 4) ‘Q a

y
```

```
apl> Rz

2 4 3

apl> L#y#(yix#3 1 2)Ta

bp
eo
an
ry

lb
yi
nr
xd

do
ux
ce
kn

apl> Ry

3 4 2
```

```
apl> tix[zix]#ix
apl> (2 4 1 ‘Q z) ix[1 3 2] tix (tix ‘Q z)

 y  2 4 1  2 4 1  y

apl> tix[wix]#ix
apl> (1 2 4 ‘Q w) ix[2 1 3] tix (tix ‘Q w)

 y  1 2 4  1 2 4  y

apl> tix[yix]#ix
apl> (1 4 2 ‘Q y) ix[2 3 1] tix (tix ‘Q y)

 y  1 4 2  1 4 2  y
```

When **p** in **z#pTr** has repetitions, **z** is a “diagonal cross section” of **r**. Given a matrix **r**, **1 1Tr** selects those items whose row and column indices are the same. For higher rank arrays, items are selected that have the same index value for the axis that is repeated in **p**, e.g. **2 1 2Ts** selects items from **s** whose first and third indices match.

Monadic transpose, Tr, is equivalent to `(‘OIRRr)Tr`. In other words it reverses the order of the axes of its argument.

APL Usage

```
apl> L # a # 3 3 R I 9

1 2 3
4 5 6
7 8 9

apl> 1 1 T a

1 5 9

apl> L#b#4 4RI16

 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

apl> 1 1Tb

1 6 11 16

apl> h#'abcdefghijkl'
apl> L#h#2 3 4Rh,I12

 a  b  c  d
 e  f  g  h
 i  j  k  l

 1  2  3  4
 5  6  7  8
 9 10 11 12

apl> 1 1 1Th

 a 6
```

```
apl> Ta

1 4 7
2 5 8
3 6 9

apl> L#c#3 4RI12

1  2  3  4
5  6  7  8
9 10 11 12

apl> 1 1Tc

1 6 11

apl> d#'one' 'for' 'all'
apl> d#d, 'head' 'to' 'toe'
apl> d#d, 'ready' 'set' 'go'
apl> L#d#3 3Rd

 one   for all
 head  to  toe
 ready set go

apl> L#v#1 1Td

 one to go

apl> Rv

3

apl> ‘=v
```

```
apl> 1 1 2Th

 a b c d
 5 6 7 8

apl> 1 2 1Th

 a e  i
 2 6 10

apl> 2 1 2Th

 a  2
 e  6
 i 10

apl> Lio#0
apl> L#k#3 2 4RI24

 0  1  2  3
 4  5  6  7

 8  9 10 11
12 13 14 15

16 17 18 19
20 21 22 23

apl> Lio#1
apl> L#a#4 3R'ramoneatenet'

ram
one
ate
net

apl> Ra

4 3
```

```
2

apl> 2 2 1Th

 a 5
 b 6
 c 7
 d 8

apl> 1 2 2Th

 a f  k
 1 6 11

apl> 2 1 1Th

 a  1
 f  6
 k 11

apl> 1 0 2Tk

 0  1  2  3
 8  9 10 11
16 17 18 19

 4  5  6  7
12 13 14 15
20 21 22 23

apl> L#z#Ta

roan
ante
meet

apl> Rz

3 4
```

```
apl> L#b#2 3R(1 1)(1 2)(1 3)(2 1)(2 2)(2 3)
```

```
 1 1  1 2  1 3
 2 1  2 2  2 3

apl> Rb

2 3

apl> ‘=b

2

apl> L#c#2 3 4RI24

 1  2  3  4
 5  6  7  8
 9 10 11 12

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

apl> Rw

4 3 2
```

```
apl> L#x#Tb

 1 1  2 1
 1 2  2 2
 1 3  2 3

apl> Rx

3 2

apl> L#w#Tc

 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23

 4 16
 8 20
12 24
```

Transpose and Selective Specification

Transpose can be used for selective specification, described on page 48:

```
apl> L#p#3 3RI9

1 2 3
4 5 6
7 8 9

apl> (1 1Tp)#10 20 30
apl> p
```

```
apl> L#r#3 3R'stypieant'

sty
pie
ant

apl> (Tr)#3 3RI9
apl> r
```

```
10  2  3                          1 4 7
 4 20  6                          2 5 8
 7  8 30                          3 6 9
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Endoper; Indxsub; Litvect; Quadout;
     Reshape; Transpos;
     Aplcb out;
     #include "aplinit.h"

     out = quadout(reshape(litvect("3 3"),indxsub(9)));
/*
                                                  1 2 3
                                                  4 5 6
                                                  7 8 9
*/
     out = quadout(transpos(litvect("1 1"),out));
/*
                                                 1 5 9
*/
     endoper(out);
}
```

See Also

Assign (#), Index Generator (I), Quad Output (L), Reshape (R).

TREE.H

Included text to define structures in building AVL-balanced trees.

```
#define SUSPNDED    1 /* 1=suspended, 0=pendent */
#define NOT_WHOLE_ARRAY  2 /* see explanation below */
#define SELSPEC          4 /* Selective specification operation
  is taking place.  All references to variable names
  are resolved to arrays of indices. */
```

```
typedef struct treelist *Treelist;
typedef struct avlnode *Avlnode;
typedef struct execstk *Execstk;
  struct treelist {
    Treelist treenext; /* next treelist element      */
    Avlnode avlhdr;    /* AVL-balanced tree          */
    Aplfunc avlfun;    /* Suspended function         */
    char *avlfname;    /* Suspended function name    */
    int avlstmt;       /* Suspended stmt             */
    int indxsave;      /* 1 if indxorg localized     */
    int indxhold;      /* Old indxorg to restore     */
    int fuzzsave;      /* 1 if fuzz localized        */
    double fuzzhold;   /* Old fuzz to restore        */
    Execstk avlexec;   /* Execution stack            */
    Apltoken avltokhd; /* Input token stack          */
    int lastfun;       /* 1=last was assignment      */
                       /* 2=last user defined func.  */
    int avloff;        /* offset in stmt where error */
    int treeflag;      /* 1 = suspended (SUSPNDED)   */
                       /* 2 = NOT_WHOLE_ARRAY        */
  };
  struct avlnode {
    Avlnode left_child; /* left child pointer */
    Avlnode rite_child; /* right child pointer */
    void *avlleaf; /* value pointer - may be NULL */
    int avlbf; /* avl balance factor */
    char *avlname; /* name of node */
  };
  struct execstk {
    Execstk execnxt;    /* next on stack */
    Apltoken avlfunst;  /* operator stack */
    Apltoken avloprst;  /* operand stack */
  };
```

Notes on lastfun:

- Compute pushes a new treelist element on the avlhdr stack, calls executf to process an APL statement, saves lastfun before popping the treelist element, and assigns the saved value to lastfun in the new top-of-stack.
- Execalt calls executg to execute an APL statement, then tests for a NULL return value, indicating a niladic defined function, and sets

lastfun to 2 in top-of-stack if so.

- Execdyan, which is called to do dyadic functions, sets lastfun to 2 in top-of-stack if it's a user defined function.
- Execexec pushes a new element on execstk, initializes $treehdr->lastfun$ to 0, then calls execexed to process an APL expression.
- Execexed, before processing each function/operator token, if it's not BASE-NULL (compute), sets $treehdr->lastfun$ to its token_code before calling execspec to do monadics and assignment.
- Execexeg saves $treehdr->lastfun$ before processing bracketed and parenthesized expressions. Then it restores the saved value to $treehdr->lastfun$.
- Execexeh tests $treehdr->lastfun$ for 2, and if so, resets it to 0. Execnext passes the value of $treehdr->lastfun$, saved before calling execexee to get the next operand, to execnexs, who tests it for LEFT_ARROW, and if so creates a list of names to its left in the expression being evaluated.
- Execspec sets $treehdr->lastfun$ to LEFT_ARROW after handling selective assignment.
- Execspex saves $treehdr->lastfun$, sets it to 0, then processes a selective specification, and restores the saved value.
- Execute initializes $treehdr->lastfun$ to 0, calls executf to process and APL expression, then tests $treehdr->lastfun$, and calls quadout to print the result if the last thing done was not LEFT_ARROW.
- Funcexee initializes $treehdr->lastfun$ to 0.
- Funcexef calls quadout if the last thing done in the most recently executed APL statement was not LEFT_ARROW.
- Functrac calls quadout if the last thing done in the most recently executed APL statement was not LEFT_ARROW.
- Slashtra initializes $treehdr->lastfun$ to 0.
- Treeroot initializes lastfun to 0 in the new treelist element.

Notes on SUSPNDED:

- Referenced in APLSI, FUNCEXEF, FUNCSUSP, and FUNCSUSQ.

- APLSI, which displays the state indicator stack, displays a star (*) next to any suspended function (treeflag == SUSPNDED) in the stack.
- FUNCEXEF, if aplerr == 997, indicating the state indicator stack is to be cleared to the next most recently suspended function, tests for treeflag & SUSPNDED to determine if it's found the next most recently suspended function.
- FUNCSUSP turns on SUSPNDED if, at initial entry, $treehdr->avlfun$ is not NULL, which means it's being called to handle terminal interaction until the user enters a right arrow to resume execution of the function being suspended. The SUSPNDED flag is turned off by FUNCSUSP and it exits under any one of three conditions:
  - 1) aplerr == 998, indicating EOF or)off.
  - 2) aplerr == 997, indicating the state stack is being cleared up to the most recently suspended function.
  - 3) FUNCSUSQ returns a 0, indicating the current function is finished.
- FUNCSUSQ, if the current statement is a right arrow, turns off SUSPNDED in the current treeflag. If the current function is suspended, it means the terminal user no longer wants it to be. He either wants the state indicator stack cleared to the next most recently suspended function, if the right arrow stands alone, or he wants execution of the suspended function to resume at the specified statement, if the right arrow doesn't stand alone.

  If the arrow stands alone, FUNCSUSQ also sets aplerr to 997 to indicate the state indicator is to be cleared to the next most recently suspended function.

Notes on NOT_WHOLE_ARRAY:

- Set on in execdyad if the current operation is assignment.
- Set on in execmonh if the current operation is not UP_ARROW.
- Referenced in execspeg, and indirectly in execspef, as follows: Execspef is called by execspez, passing an argument (nwa), which is set in execspeg to (treeflag & NOT_WHOLE_ARRAY). Execspef sets the fifth argument passed to execspeh to 1 if nwa is zero, and to $indices->aplcount$ otherwise. Execspeh expects the fifth argument to be the count of indices in a selective specification operation. If the

count is greater than 1 and matches the count of items in rite, the variable from which elements are assigned, then the individual items of rite are assigned to individual items of left. Otherwise, rite itself is assigned as an element in left, which is converted to type APLAPL if necessary.

TRIGKEYS.H

Trigkeys.h defines keys suppllied as the left argument to circular, circulax, and powerx, to identify the desired trigonometric function.

```
#define ATANH -7
#define ACOSH -6
#define ASINH -5
#define ATAN  -3
#define ACOS  -2
#define ASIN  -1
#define  TANH  7
#define  COSH  6
#define  SINH  5
#define  TAN   3
#define  COS   2
#define  SIN   1
```

USER LOAD ⎕UL **Lul**

IBM's APL2 has a system variable $\square UL$, which lists the number of users on the system. This is not implemented in CAPLIB2.

VALUE

C program usage only, returns the double floating point value of the first or only element pointed to by the aplcb passed as argument.

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Litvect; Value;
     #include "aplinit.h"
```

```
    printf("Sample usage of value = %f\n",
      value(litvect("2 3 4")));
    printf("Sample usage of value = %f\n",
      value(litvect("3.5 3 4")));
}

Sample usage of value = 2.000000
Sample usage of value = 3.500000
```

See Also

IVALUE.

)VARS

)vars [a]

Reports in alphabetical order, starting with the indicated letter if given, a list of the variables in the active workspace.

C Usage

There is no C programming interface to)*vars*.

See Also

Appendix E, Name List (Lnl).

WITHOUT ~ ~

CAPLIB2 and APL2 only, **without** is not in classic APL. The expression **z # q ∼ r** assigns to **z** the vector of items in **q** that are not in **r**. This CAPLIB2 expression always produces 1:

```
(q~r) ‘= (~qEr)/q
```

APL Usage

```
apl> 1 2 3 4 5 ~ 2 3 4     apl> 'rhyme' ~ 'myth'

1 5                        re

apl> 'go' 'to' 'it' ~ 'goto' 'it'

 go to                     apl> L#z#4 5 (I0) 6 7 ~ 9 5 3 7

                            4    6

apl> L#w#4 5 (I0) 6 7 ~ 9 5 3 7 (I0)

 4 6                       apl> Rz

apl> Rw                    3

2

apl> 3 1 4 1 5 5 ~ 3 1 4 1 5 5 ~ 4 2 5 2 6

4 5 5
```

C Usage

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
     Chrvect; Endoper; Quadout; Without;
     #include "aplinit.h"

     endoper(quadout(without(
       chrvect("rhyme"),
       chrvect("myth"))));
}

re
```

See Also

Index of (I), Member (E).

WORKSPACE

When you're using the CAPLIB2 interactive APL interpreter (**apl.exe in DOS**), all the variables and user functions you create are in main storage as a unit called the **active workspace**. An active workspace can be saved on disk using an APL System Command **)save**, and a previously saved workspace can be loaded into main storage to become the active workspace using the APL System Command **)load**. Specific variables and user functions in a workspace saved on disk can be copied into the active workspace using the APL System Command **)copy**.

See Also

Appendix E.

WORKSPACE AVAILABLE ⎕WA **Lwa**

IBM's APL2 has a system variable $\square WA$, which lists the number bytes available in the active workspace. This is not implemented in CAPLIB2.

)WRITE

)write file

The character variable **file** is written to the named DOS or Linux file. Each row of variable *file* becomes a line of text in the DOS or Linux file, delimited by a newline character ($\backslash n$ in C).

)WRITE isn't standard APL usage. I added it because I don't like the line editor offered by standard APL, preferring to use my own. Suppose I want to change function *myfunc*. I'll enter APL, issue the commands

```
file # Lcr ‘ myfunc ‘
)write myfunc.txt
)off
```

This puts me at a DOS or Linux prompt, where I'll invoke a text editor to change the function. Later I'll use **)read** (see page 273) to read the altered function back into the APL workspace.

See Also

Canonical Representation (Lcr), Defined Functions,)READ.

)WSID

)wsid <wsname>

If)wsid is issued with no operand, the name of the active workspace is listed.

If an operand is specified, the active workspace is assigned that name. A subsequent)save command without an operand causes the active workspace to be saved to disk with this name.

See Also

Appendix E.

Appendix A

ERROR MESSAGES

- 1 - Non empty variables of dissimilar type are invalid to dyadic.
- 2 - Arguments to catenate aren't conformable.
- 3 - Invalid axis in catenate.
- 4 - invalid left argument to compress/expand derived function.
- 5 - Arguments to mdivide must have ranks <= 2.
- 6 - Number of rows m/b >= number of columns in right argument to mdivide and matinv.
- 7 - Row dimension must match in the two arguments to mdivide.
- 8 - Right argument to mdivide or matinv isn't invertible.
- 9 - Invalid axis value
- 10 - Left operand to reshape must be of rank 0 or 1.
- 11 - Can't reshape something from nothing.
- 12 - Invalid constant found by litvect.
- 13 - An empty vector and undefined identity in reduce.
- 14 - The operands to format must be numeric.
- 15 - The left argument to take or drop can't have rank > 1.
- 16 - In take or drop, the left's count must equal right's rank.

- 17 - Dyadic function operands are not conformable.
- 18 - Character-to-real conversion isn't supported.
- 19 - Character-to-integer conversion isn't supported.
- 20 - An arithmetic (+-/) or math (sin, cos) function failed.
- 21 - Conversion from numeric to character is not supported.
- 22 - Ivalue has been passed an empty variable.
- 23 - Value has been passed an empty variable.
- 24 - Dimensions must conform in inner product and decode.
- 25 - The left argument to dyadic format has the wrong rank.
- 26 - The left argument to format has incorrect length.
- 27 - Out of main storage.
- 28 - A width is too large in format.
- 29 - A width is too small in format.
- 30 - No match, count of nonzero left items and right's axis in expand.
- 31 - No match, left count and right's axis in expand.
- 32 - No match, left count and right's rank in transpose.
- 33 - Left is missing a positive integer from 0 to n in transpose.
- 34 - Index out of range.
- 35 - Bad number of replacements in indexv.
- 36 - Invalid argument to factorial.
- 37 - Argument to iroll is less than 0.
- 38 - Integer division by 0.
- 39 - Real division by 0.
- 40 - Unsupported values to binom function.
- 41 - Arguments to deal must be 1 item long.
- 42 - Arguments to deal must be positive integers.

- 43 - Left must be <= right in deal.
- 44 - Null operand to quadout or execdyam.
- 45 - Argument to indexgen must be a positive integer.
- 46 - (Not currently in use).
- 47 - Argument to grade can't be character.
- 48 - Right argument to rotate has bad shape.
- 49 - Shapes of left and right excluding axis must match in rotate.
- 50 - Shapes of left and right must match in laminate.
- 51 - Axis must be positive in laminate.
- 52 - Axis too large in laminate.
- 53 - Wrong number of indices to indexm
- 54 - Index out of range in indexm.
- 55 - Error in assign.
- 56 - Argument to execute has rank greater than one.
- 57 - Argument to execute must be character.
- 58 - Syntax error.
- 59 - Operand not found where expected during execute.
- 60 - Missing result during execute.
- 61 - Stack not empty at end of expression in execute.
- 62 - Invalid result from execute.
- 63 - Missing operator during execute processing.
- 64 - Unexpected bracketed value.
- 65 - Unknown operator found during dyadic execute processing.
- 66 - Left operand to assignment must be a variable name.
- 67 - Missing monadic operand during execute processing.
- 68 - Scalar can't be indexed.

- 69 - Unknown operand type during execute.
- 70 - Missing right operand during execute.
- 71 - NULL operand during monadic processing in module execmonb.
- 72 - Invalid scalar function used with reduce or scan.
- 73 - Unknown monadic operator.
- 74 - Unexpected function operand to dot product.
- 75 - Only equal/not equal can be applied to character data.
- 76 - Scan does not accept a left argument.
- 77 - Apl null (jot) must be succeeded by a period.
- 78 - NULL output from dot product.
- 79 - Missing scalar function to the left of reduce/scan operator.
- 80 - Left bracket not found when expected in execute.
- 81 - Semicolon not found when expected in execute.
- 82 - Token expected to the left of left bracket.
- 83 - Axis must be a one-element vector or scalar.
- 84 - Artanh argument > 1.
- 85 - Left value to circular function must be integer -7 thru 7.
- 86 - Denominator of exponent must be odd if left negative to power.
- 87 - Exponent must approximate rational number if left negative to power.
- 88 - System variable not yet implemented.
- 89 - Axis indicated where it's not accepted.
- 90 - Input to quad fx m/b character matrix, or nested vector.
- 91 - Can't assign to a label.
- 92 - Variable size is too big for this implementation.
- 93 - Invalid syntax in function header.

- 94 - User function without result must stand alone.
- 95 - Invalid syntax, right arrow then cap-null in function.
- 96 - A local name can't be repeated in a function header.
- 97 - Local variable names and labels can't be synonyms in a function.
- 98 - No returned value from user function when exected.
- 99 - Unknown command.
- 100 - File doesn't exist.
- 101 - File can't be created.
- 102 - File can't be opened.
- 103 - File can't be written to.
- 104 - File can't be closed.
- 105 - File can't be read from.
- 106 - Workspace version not supported.
- 107 -)Save specifies an existing workspace which is not the current.
- 108 -)Save where?.
- 109 - Input to quadcr m/b a non-empty character vector.
- 110 - Input to quadcx or quadnc m/b character vector or matrix.
- 111 - Left argument to quadnl m/b character vector.
- 112 - Bad number of replacement values in indexm.
- 113 - Left argument to reshape must be zero or positive.
- 114 - DEL editor expected a function name.
- 115 - DEL editor found function name already in use.
- 116 - DEL editor didn't find named function to edit.
- 117 - Quad NL accepts only values 1, 2, and 3 in the right argument.
- 118 - Not grouped, name in use..
- 119 - Group name is undefined.

- 120 - No function is currently suspended to branch in.
- 121 - Left argument to replicate may contain only positive values.
- 122 - Ranks differ in disclose.
- 123 - Invalid axes.
- 124 - Arguments inconsistent in pick.
- 125 - Nested variable is argument to integer().
- 126 - Bad left argument to n-wise.
- 127 - Item out of place.
- 128 - Length error.
- 129 - Rank error.
- 130 - Argument of comples gamma function too close to a pole.
- 131 - Invalid data type for Format by Example.
- 132 - Invalide model for Format by Example.
- 133 - Domain error.
- 134 - Items of right to form must be simple numeric scalar or character vector.
- 135 - Partition needs left to be simple vector of nonnegative integers.
- 995 - Matrix divide of complex numbers not yet supported.
- 996 - Non-integer powers of complex numbers not yet supported.
- 997 - Clearing state indicator to last suspended function.
- 998 - End of APL processing.
- 999 - Internal error

Appendix B

KEYBOARD MAPPING

Aplchar is an array of character pointers that maps APL symbols to the PC keyboard. This appendix contains a table listing what **aplchar**[**n**] points to for each index value **n**. Also listed in the rightmost column are the names **#define**d in **aplchdef.h** for each index value of **aplchar**. For example, **aplchdef.h** contains the line

#define LEFT_ARROW 3

Here's a program illustrating the use of **aplchar** and **aplchdef.h**:

```
#define INCLUDES APLCB+APLCHDEF+APLMAIN+STDIO
#include "includes.h"
main()
{
   #include "aplinit.h"
   printf("Key '%s' for the APL circle slope symbol.\n",
     aplchar[CIRCLE_SLOPE]);
   printf("Key '%s' for the APL downstile symbol.\n",
     aplchar[DOWNSTILE]);
}
```

The program would produce these two lines of output:
Key 'T' for the APL circle slope symbol.
Key 'D' for the APL downstile.

APL Symbol Set	
α alpha	logamma (No APL2 symbol)
∧ and (up caret)	⍲ nand (up caret tilde)
− bar *minus*	⍱ nor (down caret tilde)
⊥ base (down tack) *decode*	≠ not equal *exclusive or*
⍎ base null (down tack jot) *execute*	≤ not greater
∩ cap (up shoe)	≥ not less
⍝ cap null (up shoe jot) *comment*	∘ null, jot *inner/outer products*
○ circle sin / cos / sinh / ⋯	ω omega
⊖ circle bar *reverse*	∨ or (down caret)
⍉ circle slope *transpose*	¯ overbar *negative numbers*
⌽ circle stile *reverse/rotate*	+ plus *plus/conjugate*
∪ cup (down shoe)	⎕ quad
∇ del	? query
⍒ del stile *grade down*	' quote
⍫ del tilde	! quote-dot *binomial/factorial*
∆ delta	⍞ quote-quad
⍋ delta stile *grade up*	ρ rho *reshape/shape*
¨ dieresis *each*	→ right arrow *branch*
÷ divide *divide/reciprocal*	] right bracket *axis/indexing*
⌹ domino (quad divide) *matrix divide/invert*	) right paren
· dot *inner/outer products*	⊃ right shoe
↓ down arrow *drop*	; semicolon
⌊ downstile *floor/minimum*	⌷ squad *index*
ε epsilon *membership*	/ slash *compress/reduce*
⍷ epsilon underbar *find*	⌿ slash bar *compress/reduce*
= equal	\ slope *expand/scan*
≡ equal underbar *depth*	⍀ slope bar *expand/scan*
> greater	⋆ star *exponential/power*
⌶ I-beam	| stile *magnitude/residue*
⍳ iota *index of/generator*	∼ tilde *not*
← left arrow *assignment*	× times *signum/times*
[left bracket	⊤ top (up tack) *encode*
(left parenthesis	⍕ top null (up tack jot) *format*
⊂ left shoe	_ underbar
< less	↑ up arrow *take*
⍟ log (circle star) *logarithms*	⌈ upstile *ceiling, maximum*

Table B.1: APL Symbol Set

CAPLIB^2 Symbol Set	
‘A alpha	‘2 logamma
& and (up caret)	P nand (up caret tilde)
_ bar *minus*	‘V nor (down caret tilde)
B base (down tack) *decode*	^= not equal *exclusive or*
C base null (down tack jot) *execute*	K not greater
‘C cap (up shoe)	Z not less
" cap null (up shoe jot) *comment*	J null, jot *inner/outer products*
O circle sin / cos / sinh ···	‘W omega
‘R circle bar *reverse*	V or (down caret)
T circle slope *transpose*	− overbar *negative numbers*
‘O circle stile *reverse/rotate*	+ plus *plus/conjugate*
‘U cup (down shoe)	L quad
G del	? query
W del stile *grade down*	‘ quote
‘G del tilde	! quote-dot *binomial/factorial*
H delta	Q quote-quad
A delta stile *grade up*	R rho *reshape/shape*
‘1 dieresis *each*	$ right arrow *branch*
% divide *divide/reciprocal*	] right bracket *axis/indexing*
M domino (quad divide) *matrix divide/invert*	) right paren
. dot *inner/outer products*	‘X right shoe
U down arrow *drop*	; semicolon
D downstile *floor/minimum*	‘Q squad *index*
E epsilon *membership*	/ slash *compress/reduce*
‘E epsilon underbar *find*	‘/ slash bar *compress/reduce*
= equal	\ slope *expand/scan*
‘= equal underbar *depth*	‘\ slope bar *expand/scan*
> greater	* star *exponential/power*
‘I I-beam	| stile *magnitude/residue*
I iota *index of/generator*	~ tilde *not*
# left arrow *assignment*	X times *signum/times*
[left bracket	N top (up tack) *encode*
(left parenthesis	F top null (up tack jot) *format*
‘Z left shoe	‘_ underbar
< less	Y up arrow *take*
@ log (circle star) *logarithms*	S upstile *ceiling, maximum*

Table B.2: CAPLIB^2 Symbol Set

APLCHAR Array Content				
Index	Keys	ASCII	Description	#Defined Name
0		32	**Space**	SPACE
1	!	33	**Quote dot**	QUOTE_DOT
			binomial/factorial	
2	”	34	**Cap null**	CAP_NULL
			comment	
3	#	35	**Left arrow**	LEFT_ARROW
			assignment	
4	$	36	**Right arrow**	RIGHT_ARROW
			branch	
5	%	37	**Divide**	DIVIDE
6	&	38	**And**	AND
7	’	39	**Quote**	QUOTE
8	(	40	**Left paren**	LEFT_PAREN
9	)	41	**Right paren**	RIGHT_PAREN
10	*	42	**Star**	STAR
			exponential/power	
11	+	43	**Plus**	PLUS
			plus/conjugate	
12	,	44	**Comma**	COMMA
			catenate/laminate/ravel	
13	-	45	**Overbar**	OVERBAR
			negative numbers	
14	.	46	**Dot**	DOT
			inner/outer product	
15	/	47	**Slash**	SLASH
			compress/reduction	
16	0	48	**Digit 0**	APL_0
17	1	49	**Digit 1**	APL_1
18	2	50	**Digit 2**	APL_2
19	3	51	**Digit 3**	APL_3
20	4	52	**Digit 4**	APL_4
21	5	53	**Digit 5**	APL_5
22	6	54	**Digit 6**	APL_6
23	7	55	**Digit 7**	APL_7
24	8	56	**Digit 8**	APL_8
25	9	57	**Digit 9**	APL_9

Table B.3: APLCHAR Array Content, Part 1 of 5

APLCHAR Array Content				
Index	Keys	ASCII	Description	#Defined Name
26	:	58	**Colon**	COLON
27	;	59	**Semicolon**	SEMICOLON
28	<	60	**Less**	LESS
29	=	61	**Equal**	EQUAL
30	>	62	**Greater**	GREATER
31	?	63	**Query** deal/roll	QUERY
32	@	64	**Log** logarithm	LOG
33	A	65	**Delta stile** grade up	DELTA_STILE
34	B	66	**Base** decode	BASE
35	C	67	**Base null** execute	BASE_NULL
36	D	68	**Downstile** floor/minimum	DOWNSTILE
37	E	69	**Epsilon** membership	EPSILON
38	F	70	**Top null** format	TOP_NULL
39	G	71	**Del**	DEL
40	H	72	**Delta**	DELTA
41	I	73	**Iota** index generator/index of	IOTA
42	J	74	**Null** inner/outer product	APL_NULL
43	K	75	**Not greater**	NOT_GREATER
44	L	76	**Quad** input/output	QUAD
45	M	77	**Domino** matrix divide	DOMINO
46	N	78	**Top** encode	TOP

Table B.4: APLCHAR Array Content, Part 2 of 5

APLCHAR Array Content				
Index	Keys	ASCII	Description	#Defined Name
47	O	79	**Circle** circular	CIRCLE
48	P	80	**Nand**	NAND
49	Q	81	**Quote-quad** character input	QUOTE_QUAD
50	R	82	**Rho** reshape/shape	RHO
51	S	83	**Upstile** ceiling/maximum	UPSTILE
52	T	84	**Circle slope** transpose	CIRCLE_SLOPE
53	U	85	**Down arrow** drop	DOWN_ARROW
54	V	86	**Or**	OR
55	W	87	**Del stile** grade down	DEL_STILE
56	X	88	**Times** signum/times	TIMES
57	Y	89	**Up arrow** take	UP_ARROW
58	Z	90	**Not less**	NOT_LESS
59	[	91	**Left bracket**	LEFT_BRACKET
60	\	92	**Slope** expand/scan	SLOPE
61	]	93	**Right bracket**	RIGHT_BRACKET
62	^ =	94 61	**Not equal** exclusize or	NOT_EQUAL
63	_	95	**Bar** minus/negative	BAR
64	‘/	96 47	**Slash bar** compress/reduce	SLASH_BAR
65	‘1	96 49	**Dieresis**	DIERESIS
66	‘A	96 65	**Alpha**	ALPHA
67	‘C	96 67	**Cap**	CAP
68	‘G	96 71	**Del tilde**	DEL_TILDE

Table B.5: APLCHAR Array Content, Part 3 of 5

APLCHAR Array Content				
Index	Keys	ASCII	Description	#Defined Name
69	‘I	96 73	**I-beam**	I_BEAM
70	‘O	96 79	**Circle stile**	CIRCLE_STILE
			reverse/rotate	
71	‘R	96 82	**Circle bar**	CIRCLE_BAR
			reverse	
72	‘U	96 85	**Cup**	CUP
73	‘V	96 86	**Nor**	NOR
74	‘W	96 87	**Omega**	OMEGA
75	‘X	96 88	**Right shoe**	RIGHT_SHOE
76	‘Z	96 90	**Left shoe**	LEFT_SHOE
77	‘\	96 92	**Slope bar**	SLOPE_BAR
			expand/scan	SLOPE_BAR
78	‘_	96 95	**Underbar**	UNDERBAR
79	‘a	96 97	**A underscore**	APL_A_UNDERSCORE
80	‘b	96 98	**B underscore**	APL_B_UNDERSCORE
81	‘c	96 99	**C underscore**	APL_C_UNDERSCORE
82	‘d	96 100	**D underscore**	APL_D_UNDERSCORE
83	‘e	96 101	**E underscore**	APL_E_UNDERSCORE
84	‘f	96 102	**F underscore**	APL_F_UNDERSCORE
85	‘g	96 103	**G underscore**	APL_G_UNDERSCORE
86	‘h	96 104	**H underscore**	APL_H_UNDERSCORE
87	‘i	96 105	**I underscore**	APL_I_UNDERSCORE
88	‘j	96 106	**J underscore**	APL_J_UNDERSCORE
89	‘k	96 107	**K underscore**	APL_K_UNDERSCORE
90	‘l	96 108	**L underscore**	APL_L_UNDERSCORE
91	‘m	96 109	**M underscore**	APL_M_UNDERSCORE
92	‘n	96 110	**N underscore**	APL_N_UNDERSCORE
93	‘o	96 111	**O underscore**	APL_O_UNDERSCORE
94	‘p	96 112	**P underscore**	APL_P_UNDERSCORE
95	‘q	96 113	**Q underscore**	APL_Q_UNDERSCORE
96	‘r	96 114	**R underscore**	APL_R_UNDERSCORE
97	‘s	96 115	**S underscore**	APL_S_UNDERSCORE
98	‘t	96 116	**T underscore**	APL_T_UNDERSCORE
99	‘u	96 117	**U underscore**	APL_U_UNDERSCORE
100	‘v	96 118	**V underscore**	APL_V_UNDERSCORE
101	‘w	96 119	**W underscore**	APL_W_UNDERSCORE
102	‘x	96 120	**X underscore**	APL_X_UNDERSCORE
103	‘y	96 121	**Y underscore**	APL_Y_UNDERSCORE
104	‘z	96 122	**Z underscore**	APL_Z_UNDERSCORE

Table B.6: APLCHAR Array Content, Part 4 of 5

APLCHAR Array Content				
Index	Keys	ASCII	Description	#Defined Name
105	a	97	**A**	APL_A
106	b	98	**B**	APL_B
107	c	99	**C**	APL_C
108	d	100	**D**	APL_D
109	e	101	**E**	APL_E
110	f	102	**F**	APL_F
111	g	103	**G**	APL_G
112	h	104	**H**	APL_H
113	i	105	**I**	APL_I
114	j	106	**J**	APL_J
115	k	107	**K**	APL_K
116	l	108	**L**	APL_L
117	m	109	**M**	APL_M
118	n	110	**N**	APL_N
119	o	111	**O**	APL_O
120	p	112	**P**	APL_P
121	q	113	**Q**	APL_Q
122	r	114	**R**	APL_R
123	s	115	**S**	APL_S
124	t	116	**T**	APL_T
125	u	117	**U**	APL_U
126	v	118	**V**	APL_V
127	w	119	**W**	APL_W
128	x	120	**X**	APL_X
129	y	121	**Y**	APL_Y
130	z	122	**Z**	APL_Z
131	|	124	**Stile** magnitude/residue	STILE
132	~	126	**Tilde** not	TILDE
133	‘=	96 61	**Equal Underbar** depth/match	EQUAL_UNDERBAR
134	‘E	96 69	**Epsilon Underbar** find	EPSILON_UNDERBAR
135	‘Q	96 81	**Squad** index	SQUAD
136	‘2	96 50	**Logamma** logamma	LOGAMMA

Table B.7: APLCHAR Array Content, Part 5 of 5

Appendix C

MIXED FUNCTIONS

In classic APL, **Mixed functions** could easily be grouped into five classes. APL2 requires a sixth group:

- **Nesting: Concerning the Nesting of Arrays (APL2 only)**
  - **Depth:** Reports level of nesting (APL2 only).
  - **Disclose:** Removes a level of nesting (APL2 only).
  - **Enclose:** Adds a level of nesting (APL2 only).
  - **Enlist:** Removes nesting, creating a simple vector (APL2 only).
  - **Partition:** Adds a level of nesting (APL2 only).
- **Structured: Concerning the Structure of Arrays**
  - **Catenate:** Append one argument to another, retaining rank.
  - **Laminate:** Append one argument to another, increasing rank.
  - **Ravel:** Make its argument into a vector.
  - **Reshape:** Reshapes its argument.
  - **Reverse:** Reverse the order of items in its argument.
  - **Rotate:** Rotate items within its argument.
  - **Shape:** Gives the shape of its argument.
  - **Transpose:** Shuffle dimensions about.
- **Selection: Concerning Selection from Arrays**
  - **Compress:** Eliminate selected items according to a pattern.

  - **Expand:** Insert zeros or blanks according to a pattern.
  - **Drop:** Drop selected items from its argument.
  - **First:** Selects the first item (APL2 only).
  - **Indexing:** Select items according to numeric position (index).
  - **Pick:** Select an item by "path indices" (APL2 only).
  - **Take:** Take selected items from its argument.
  - **Without:** Yields items in L that do not occur in R (APL2 only).

- **Generator: Concerning Generation of Selector Information**
  - **Deal:** Random number generation.
  - **Find:** Finds patterns (APL2 only).
  - **Grade:** Numeric positions (indexes) of items in ascending/descending order.
  - **Index Generator:** Incremental sequence of integers.
  - **Index Of:** Numeric position (index) of selected items.
  - **Match:** Indicates if arguments are the same in structure and data (APL2 only).
  - **Membership:** Indicate whether items or one argument are members of another.

- **Numerical: Concerning Numerical Calculations**
  - **Decode:** Base number transformations.
  - **Encode:** Base number transformations.
  - **Matrix Divide.**
  - **Matrix Inverse.**

- **Transformation: Concerning Data Transformation**
  - **Execute:** Execute character vectors as APL statements.
  - **Format:** Format numeric data for display.

Appendix D

SCALAR FUNCTIONS

Scalar functions accept scalar arguments and return scalar values. They can be categorized monadic or dyadic (one argument or two), and by the data type of their arguments and returned value (integer, real, or character). A scalar is a single element, as opposed to an array. The tables that follow show all the scalar functions supported by CAPLIB2.

Scalar Functions, Monadic - Real Argument, Real Result

$abs(-3.14) \rightarrow 3.14.$
$ceil(3.14) \rightarrow 4.0$, $ceil(-3.14) \rightarrow -3.0.$
$conjug(23.0) \rightarrow 23.0$, $conjug(-23.0) \rightarrow -23.0$
$factorl(0.5) \rightarrow .8862\ldots$, $factorl(5.0) \rightarrow 120.0$
$floor(3.14) \rightarrow 3.00$, $floor(-3.14) \rightarrow -4.00$
$negative(-2.0) \rightarrow 2.0$, $negative(2.0) \rightarrow -2.0$
$pitimes(1) \rightarrow 3.14\ldots$, $pitimes(2) \rightarrow 6.28\ldots$
$recip(4) \rightarrow 0.25$
$signdbl(2.0) \rightarrow 1.0$, $signdbl(-2.0) \rightarrow -1.0$, $signdbl(0.0) \rightarrow 0.0$

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
   Ceil; Dabs; Endoper; Floor; Litvect; Monadic;
   Negative; Quadout; Recip; Signdbl;
   #include "aplinit.h"

   endoper(quadout(monadic(dabs,
```

```
     litvect("3.14 -3.14")))); /* -> 3.14 3.14 */

   endoper(quadout(monadic(ceil,
     litvect("3.14 -3.14")))); /* -> 4 -3 */

   endoper(quadout(monadic(floor,
     litvect("3.14 -3.14")))); /* -> 3 -4 */

   endoper(quadout(monadic(negative,
     litvect("3.14 -3.14")))); /* -> -3.14 3.14 */

   endoper(quadout(monadic(recip,
     litvect("1 2 3 4")))); /* ----> 1 0.5 0.333 0.25 */

   endoper(quadout(monadic(signdbl,
     litvect("3.14 0 -3.14")))); /* --> 1 0 -1  */
}

#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
   Endoper; Exp; Factorl; Litvect; Log;
   Monadic; Pitimes; Quadout;
   #include "aplinit.h"

   endoper(quadout(monadic(exp,
     litvect("0 1 2 3 4"))));

 /* 1 2.7182818 7.389056 20.085537 54.59815 */

   endoper(quadout(monadic(factorl,
     litvect("0 1 2 3 4 -0.5 0.5 1.5"))));

 /* 1 1 2 6 24 1.7724539 0.8862269 1.3293404 */

   endoper(quadout(monadic(log,
     litvect("1 2 3 4"))));

 /* 0 0.6931472 1.0986123 1.3862944 */

   endoper(quadout(monadic(pitimes,
```

```
     litvect("0 1 2 3 4"))));

 /* 0 3.1415927 6.2831853 9.4247780 12.56637 */
}
```

Scalar Functions, Monadic - Integer Argument, Integer Result

$iabs(-3) \to 3$, $iabs(3) \to 3$
$iconjug(3) \to 3$, $iconjug(-3) \to -3$
$ifactorl(5) \to 120$
$iroll(6) \to 5$ (random choice)
$inegativ(-2) \to 2$, $inegative(2) \to -2$
$inot(0) \to 1$, $inot(1) \to 0$, $inot(5) \to 0$
$isign(2) \to 1$, $isign(0) \to 0$, $isign(-2) \to -1$

```
/* integer scalar monadic exercises */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
main()
{
   struct aplcb *litvect(),*imonadic(),*quadout();
   int iabs(),ifactorl(),iroll(),inegativ(),inot(),isign();
   #include "aplinit.h"

   endoper(quadout(imonadic(iabs,
      litvect("3 -3")))); --------> 3 3

   endoper(quadout(imonadic(ifactorl,
      litvect("0 1 2 3 4")))); ---> 1 1 2 6 24

   endoper(quadout(imonadic(iroll,
      litvect("5 5 5 5 5")))); ---> 1 3 1 2 1

   endoper(quadout(imonadic(inegativ,
      litvect("3 -3")))); --------> -3 3

   endoper(quadout(imonadic(inot,
      litvect("0 1 2 3 4")))); ---> 1 0 0 0 0

   endoper(quadout(imonadic(isign,
      litvect("3 0 -3")))); ------> 1 0 -1
```

```
}
```

Scalar Functions, Monadic - Real Argument, Integer Result

$not(0.0) \to 1$, $not(1.0) \to 0$, $not(5.0) \to 0$
$signum(2.1) \to 1$, $signum(0.0) \to 0$, $signum(-2.5) \to -1$

```
/* real scalar monadic exercises */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
main()
{
     struct aplcb *litvect(),*rmonadic(),*quadout();
     double not(),signum();
     #include "aplinit.h"

   endoper(quadout(rmonadic(not,
      litvect("0 1.1 2.2 3.3")))); ---> 1 0 0 0

   endoper(quadout(rmonadic(signum,
      litvect("3.14 0 -3.14")))); ----> 1 0 -1
}
```

Scalar Functions, Dyadic - Real Arguments, Real Result

$binom(2.0, 5.0) \to 10.0$, $binom(2.1, 5.6) \to 13.484\ldots$
$divide(21.0, 2.0) \to 10.5$
$logrithm(10.0, 100.0) \to 2.0$, $logrithm(2.0, 32.0) \to 5$
$max(3.0, 7.0) \to 7.0$
$min(3.0, 7.0) \to 3.0$
$minus(2.0, 3.2) \to -1.2$
$plus(2.0, 3.2) \to 5.2$
$power(2.0, 5.0) \to 32.0$
$residue(1.0, 2.385) \to 0.385$, $residue(0.0, 2.385) \to 2.385$
$times(2.0, 3.2) \to 6.4$

```
/* Real dyadic operations */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
```

```
main()
{
   struct aplcb *dyadic(),*litvect(),*quadout();
   double binom(),divide(),logrithm(),max(),
        min(),minus(),plus(),
        power(),residue(),times();
   #include "aplinit.h"

   endoper(quadout(dyadic(binom,
      litvect("2 2.1"),
      litvect("5 5.6")))); -----------> 10 13.4848711456

   endoper(quadout(dyadic(divide,
      litvect("0.0 1.0 1.0 1.0"),
      litvect("0.0 1.0 2.0 3.0")))); -> 1 1 0.5 0.333333

   endoper(quadout(dyadic(logrithm,
      litvect(" 2.0  4.0 10.0"),
      litvect("16.0 16.0 16.0")))); --> 4 2 1.2041199827

   endoper(quadout(dyadic(max,
      litvect("-3.1 3.1 3.1"),
      litvect("-2.4 5.1 0.0")))); ----> -2.4 5.1 3.1

   endoper(quadout(dyadic(min,
      litvect("-3.1 3.1 3.1"),
      litvect("-2.4 5.1 0.0")))); -----> -3.1 3.1 0

   endoper(quadout(dyadic(minus,
      litvect("-3.1 3.1 3.1"),
      litvect("-2.4 5.1 0.0")))); -----> -0.7 -2 3.1

   endoper(quadout(dyadic(plus,
      litvect("-3.1 3.1 3.1"),
      litvect("-2.4 5.1 0.0")))); -----> -5.5 8.2 3.1

   endoper(quadout(dyadic(power,
      litvect("2.0 2.0 2.0 10.0"),
      litvect("0.0 1.0 2.0  2.0")))); -> 1 2 4 100

   endoper(quadout(dyadic(residue,
      litvect(" 0.0 1.0  "),
```

```
      litvect("-3.0 2.385")))); -------> -3 0.385

   endoper(quadout(dyadic(times,
      litvect("2.0 2.0 2.0"),
      litvect("0.0 1.0 3.0")))); ------> 0 2 6
}
```

Scalar Functions, Dyadic - Integer Arguments, Integer Result

$ibinom(2,5) \rightarrow 10$
$idivide(21,2) \rightarrow 10$
$imax(3,7) \rightarrow 7$
$imin(3,7) \rightarrow 3$
$iminus(2,3) \rightarrow -1$
$iplus(2,3) \rightarrow 5$
$iresidu(2,21) \rightarrow 1$
$itimes(3,7) \rightarrow 21$

```
/* integer dyadic operations */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
main()
{
   struct aplcb *idyadic(),*litvect(),*quadout();
   int ibinom(),idivide(),imax(),imin(),iminus(),iplus(),
        iresidu(),itimes();
   #include "aplinit.h"

   endoper(quadout(idyadic(ibinom,
     litvect("2 5 3 -2  2  2"),
     litvect("5 2 3 -2 -2 -5")))); ------> 10 0 1 1 3 15

   endoper(quadout(idyadic(idivide,
     litvect("-3 -2 -1 0  1  2  3"),
     litvect("-3  2  1 0 -1 -2  3")))); -> 1 -1 -1 1 -1 -1 1

   endoper(quadout(idyadic(imax,
     litvect("-3 -2 -1 0  1  2  3"),
     litvect("-3  2  1 0 -1 -2  3")))); -> -3 2 1 0 1 2 3

   endoper(quadout(idyadic(imin,
```

```
     litvect("-3 -2 -1 0  1  2  3"),
     litvect("-3  2  1 0 -1 -2  3")))); -> -3 -2 -1 0 -1 -2 3

   endoper(quadout(idyadic(iminus,
     litvect("-3 -2 -1 0  1  2  3"),
     litvect("-3  2  1 0 -1 -2  3")))); -> 0 -4 -2 0 2 4 0

   endoper(quadout(idyadic(iplus,
     litvect("-3 -2 -1 0  1  2  3"),
     litvect("-3  2  1 0 -1 -2  3")))); -> -6 0 0 0 0 0 6

   endoper(quadout(idyadic(iresidu,
     litvect(" 0  2  2  2 -2 -2"),
     litvect("-3  8  3 -3  3 -3")))); ---> -3 0 1 1 -1 -1

   endoper(quadout(idyadic(itimes,
     litvect("-3 -2 -1 0  1  2  3"),
     litvect("-3  2  1 0 -1 -2  3")))); -> 9 -4 -1 0 -1 -4 9
}
```

Scalar Relational Functions - Real Arguments, Integer Result

$eq(2.0, 3.0) \to 0$, $eq(3.0, 3.0) \to 1$, $eq(4.0, 3.0) \to 0$
$ge(2.0, 3.0) \to 0$, $ge(3.0, 3.0) \to 1$, $ge(4.0, 3.0) \to 1$
$gt(2.0, 3.0) \to 0$, $gt(3.0, 3.0) \to 0$, $gt(4.0, 3.0) \to 1$
$le(2.0, 3.0) \to 1$, $le(3.0, 3.0) \to 1$, $le(4.0, 3.0) \to 0$
$lt(2.0, 3.0) \to 1$, $lt(3.0, 3.0) \to 0$, $lt(4.0, 3.0) \to 0$
$ne(2.0, 3.0) \to 1$, $ne(3.0, 3.0) \to 0$, $ne(4.0, 3.0) \to 1$

```
/* Relational functions - real arguments, integer results */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
main()
{
   struct aplcb *litvect(),*quadout(),*relative();
   int eq(),ge(),gt(),le(),lt(),ne();
   #include "aplinit.h"

   endoper(quadout(relative(eq,
      litvect("2.0 3.0 4.0"),
      litvect("3.0 3.0 3.0")))); --> 0 1 0
```

```
   endoper(quadout(relative(ge,
      litvect("2.0 3.0 4.0"),
      litvect("3.0 3.0 3.0")))); --> 0 1 1

   endoper(quadout(relative(gt,
      litvect("2.0 3.0 4.0"),
      litvect("3.0 3.0 3.0")))); --> 0 0 1

   endoper(quadout(relative(le,
      litvect("2.0 3.0 4.0"),
      litvect("3.0 3.0 3.0")))); --> 1 1 0

   endoper(quadout(relative(lt,
      litvect("2.0 3.0 4.0"),
      litvect("3.0 3.0 3.0")))); --> 1 0 0

   endoper(quadout(relative(ne,
      litvect("2.0 3.0 4.0"),
      litvect("3.0 3.0 3.0")))); --> 1 0 1
}
```

Scalar Relational Functions - Integer Arguments, Integer Result

$$\boxed{\begin{array}{l} ieq(2,3) \rightarrow 0,\ ieq(3,3) \rightarrow 1,\ ieq(4,3) \rightarrow 0 \\ ige(2,3) \rightarrow 0,\ ige(3,3) \rightarrow 1,\ ige(4,3) \rightarrow 1 \\ igt(2,3) \rightarrow 0,\ igt(3,3) \rightarrow 0,\ igt(4,3) \rightarrow 1 \\ ile(2,3) \rightarrow 1,\ ile(3,3) \rightarrow 1,\ ile(4,3) \rightarrow 0 \\ ilt(2,3) \rightarrow 1,\ ilt(3,3) \rightarrow 0,\ ilt(4,3) \rightarrow 0 \\ ine(2,3) \rightarrow 1,\ ine(3,3) \rightarrow 0,\ ine(4,3) \rightarrow 1 \end{array}}$$

```
/* Relational functions - integer arguments and results */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
main()
{
   struct aplcb *litvect(),*quadout(),*idyadic();
   int ieq(),ige(),igt(),ile(),ilt(),ine();
   #include "aplinit.h"

   endoper(quadout(idyadic(ieq,
```

```
      litvect("2 3 4"),
      litvect("3 3 3")))); --> 0 1 0

   endoper(quadout(idyadic(ige,
      litvect("2 3 4"),
      litvect("3 3 3")))); --> 0 1 1

   endoper(quadout(idyadic(igt,
      litvect("2 3 4"),
      litvect("3 3 3")))); --> 0 0 1

   endoper(quadout(idyadic(ile,
      litvect("2 3 4"),
      litvect("3 3 3")))); --> 1 1 0

   endoper(quadout(idyadic(ilt,
      litvect("2 3 4"),
      litvect("3 3 3")))); --> 1 0 0

   endoper(quadout(idyadic(ine,
      litvect("2 3 4"),
      litvect("3 3 3")))); --> 1 0 1
}
```

Scalar Boolean Functions - Real Arguments, Integer Result

	f(0.0,0.0)	f(0.0,1.0)	f(1.0,0.0)	f(1.0,1.0)
and	0	0	0	1
nand	1	1	1	0
nor	1	0	0	0
or	0	1	1	1

Real arguments to boolean functions are converted to boolean 0 or 1 depending on whether their magnitude exceeds **comparison tolerance**. Thus, $and(5.4, 2.3)$ returns 1.

```
/* Boolean functions - real arguments, integer results */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
main()
{
```

```
   struct aplcb *litvect(),*quadout(),*relative();
   int and(),nand(),nor(),or();
   #include "aplinit.h"

   endoper(quadout(relative(and,
      litvect("0.0 0.0 1.0 1.0"),
      litvect("0.0 1.0 0.0 1.0")))); --> 0 0 0 1

   endoper(quadout(relative(nand,
      litvect("0.0 0.0 1.0 1.0"),
      litvect("0.0 1.0 0.0 1.0")))); --> 1 1 1 0

   endoper(quadout(relative(nor,
      litvect("0.0 0.0 1.0 1.0"),
      litvect("0.0 1.0 0.0 1.0")))); --> 1 0 0 0

   endoper(quadout(relative(or,
      litvect("0.0 0.0 1.0 1.0"),
      litvect("0.0 1.0 0.0 1.0")))); --> 0 1 1 1
}
```

Scalar Boolean Functions - Integer Arguments, Integer Result

	f(0,0)	f(0,1)	f(1,0)	f(1,1)
iand	0	0	0	1
inand	1	1	1	0
inor	1	0	0	0
ior	0	1	1	1

Arguments to boolean functions are converted to boolean 0 or 1 depending on whether they are zero or non-zero. Thus, $iand(3,7)$ returns 1, and $iand(3,0)$ returns 0.

```
/* Boolean functions - integer arguments and results */
#include <stdio.h>
#include "aplcb.h"
#include "aplext.h"
main()
{
   struct aplcb *idyadic(),*litvect(),*quadout();
   int iand(),inand(),inor(),ior();
   #include "aplinit.h"
```

```
   endoper(quadout(idyadic(iand,
      litvect("0 0 1 1"),
      litvect("0 1 0 1")))); --> 0 0 0 1

   endoper(quadout(idyadic(inand,
      litvect("0 0 1 1"),
      litvect("0 1 0 1")))); --> 1 1 1 0

   endoper(quadout(idyadic(inor,
      litvect("0 0 1 1"),
      litvect("0 1 0 1")))); --> 1 0 0 0

   endoper(quadout(idyadic(ior,
      litvect("0 0 1 1"),
      litvect("0 1 0 1")))); --> 0 1 1 1
}
```

Appendix E

System Commands

System Commands are all prefixed with a left parenthesis. They control starting and stopping the APL interpreter, loading and saving of workspaces, and transferring data from one workspace to another. Many of them also display information about the current state of things in the active workspace. In the brief syntax descriptions below, brackets indicate items that are optional, such as the **a** in "`)fns [a]`":

- **)clear** – Clears out the active workspace. Expunges all variables and defined functions so the workspace is empty.
- **)copy wsname object** [**object**...] – The named **object**(s) are copied from the specified workspace named **wsname** on disk into the active workspace.
- **)drop wsname** – The workspace named **wsname** is deleted from disk.
- **)erase** [**object**...] – The named **object**s are expunged from the active workspace.
- **)fns** [**a**] – Displays a list of defined functions in the active workspace, starting with the optional letter indicated by [**a**].
- **)group grpname** [**grpname**] [**object**...] – The list of **object**s are grouped so they can be easily referenced by the single name **grpname**.
- **)grp grpname** – Displays the membership of the named group.
- **)grps** [**a**] – Displays the groups in the active workspace, in alphabetical order, optionally starting with the letter indicated by **a**.

- **)load wsname** – Workspace **wsname** is loaded into memory as the active workspace.
- **)off** – Leave the interactive APL interpreter, and return to DOS.
- **)pcopy wsname object** [**object**. . .] – Identical in function to)copy, except a named object will not be replaced if it already exists in the active workspace.
- **)save** [**wsname**] – Saves the active workspace to disk with file name **wsname**.apl.
- **)si** – Displays the state indicator, showing halted defined functions.
- **)sinl** – Same as)si, except local names are listed for each defined function.
- **)vars** [**a**] – Displays a list of variables in the active workspace, in alphabetical order, optionally starting with the letter indicated by **a**.
- **)wsid** [**wsname**] – If **wsname** is specified, it becomes the name of the active workspace. Otherwise, the name of the active workspace is displayed.

Appendix F

Compiling and Linking C Programs in DOS

With the DOS version of CAPLIB^2 I used the compiler and linker from Zortech Incorporated for the routines in library CAP2.LIB. Other compilers and linkers may work. C programs that invoke CAPLIB^2 routines in DOS were compiled with a command like the following:

ztc -c -ml -A sample

where:

- -c tells the compiler to compile only, don't invoke the linker.
- -ml indicates the large memory model is to be used. All CAPLIB^2 routines are compiled and linked into library **CAP2.LIB** using this option.
- -A tells the compiler to enforse strict ANSI compatibility. This is not required, but recommended.
- **Sample** is the name of the file containing the C source code to be compiled. The compiler assumes the file is named **sample.c**.

Blink is Zortech's program linker. It is invoked to combine a C program with the CAPLIB^2 subroutines from library **CAP2.LIB**. I used this command to link the sample program:

blink sample,,sample,cap/noi

where:

- **Sample** will be the name of the **.exe** file created by the linker.
- **cap2** refers to library **CAP2.LIB**, which contains all of the CAPLIB2 routines.
- **/noi** tells the linker to be case sensitive.

Sample Program:

```
#define INCLUDES APLCB+APLMAIN+STDIO
#include "includes.h"
main()
{
Chrvect; Endoper; Member; Quadout;
     #include "aplinit.h"

     endoper(quadout(member(
       chrvect("abcdefgh"),
       chrvect("coffee"))));
}

0 0 1 0 1 1 0 0
```

Appendix G

Compiling and Linking C Programs in Linux

With the Linux version of CAPLIB^2, I used the GNU gcc compiler and linker. The archive containing all the CAPLIB^2 subroutines, **cap2/lib/libcap2.a**, was built with the three shell scripts **caplib2, za,** and **lib2**, described below.

caplib2

```
# cap2/bat/caplib2
# Rebuilds $HOME/cap2/lib/libcap2.a from scratch
cp ~/cap2/include/apldebug.h0 ~/cap2/include/apldebug.h
rm $HOME/cap2/lib/libcap2.a
cd $HOME/cap2/source
for i in `ls *.c | sed 's/\.c.*//'`
do
     case $i in
          apl)      echo "Skipping $i" ;;
          apldebug) echo "Skipping $i" ;;
          mem)      echo "Skipping $i" ;;
          *)        echo "Doing $i" ; za $i ;;
     esac
done
```

za

```
# cap2/bat/za
```

```
z $1 $2 $3 $4
rc=$?
case $rc in
 0)  base=‘basename $1‘
     lib2 $base ;;
 *)  echo "bad compile = $rc"
     read ltr ;;
esac
```

lib2

```
# cap2/bat/lib2
ar rs $HOME/cap2/lib/libcap2.a $1.o
rm $1.o
```

Compile and Link Sample C programs

Sample C programs that invoke CAPLIB^{2} routines in Linux were compiled and linked using these shell scripts:

z

```
# cap2/bat/z
gcc -c -I$HOME/cap2/include \
    -I$HOME/include $2 $3 $4 $5 $6 $1.c
```

zl

```
# cap2/bat/zl
z $1
rc=$?
case $rc in
 0)  base=‘basename $1‘
     ld -dll-verbose -m486 -o $base /usr/lib/crt0.o \
          -L/home/smitty/cap2/lib \
          -L/usr/lib/gcc-lib/i486-linux/2.6.3 \
          $base.o -lcap2 -lgcc -lc -lm
     rc=$?
     rm $base.o
     ;;
 *)  echo "Bad compile = $rc"
     ;;
esac
```

```
16 exit $rc
```

Shell script **zl** must be altered to fit the version of gcc on your system. You can find out what parameters you'll need by executing a command like the following:

```
gcc -I$HOME/cap2/include -I$HOME/include \
     -v -L$HOME/cap2/lib -lcap2 -lgcc -lc -lm \
     -o sam001 $HOME/cap2/sample/c/sam001.c
```

The above won't produce a proper executable, but you can examine the output for the **ld** command and the requisite parameters:

```
...lines omitted...
 ld -dll-verbose -m486 -o sam001 /usr/lib/crt0.o \
 -L/home/smitty/cap2/lib \
 -L/usr/lib/gcc-lib/i486-linux/2.6.3 \
 -lcap2 -lc -lgcc -lm /tmp/cca001821.o -lgcc -lc -lgcc
...lines omitted...
```

Appendix H

Memory Debugging

Because CAPLIB^2 is so reliant on the dynamic allocation of memory via the **malloc** and **free** standard C library routines, bugs such as references to previously freed memory are easily introduced. The Zortech C compiler I purchased from Symantec contains a memory debugger that was indispensable in debugging these types of problems.

Written by Walter Bright, the memory debugger is contained in two source files, **mem.c** and **mem.h**. I would've liked to include them in the source for CAPLIB^2, but was unable to contact the author to obtain permission. I'll describe in this appendix how the CAPLIB^2 routines use them under Linux, and perhaps this will help if you have access to an equivalent memory debugger.

When APLDEBUG (see page 33) is added to the #define INCLUDES statement (see page 178), it causes **apldebug.h** to be included. This sets preprocessor variable APL_DEBUG to either 1 or 0, indicating whether memory debugging is to occur. There are ten CAPLIB^2 routines that are sensitive to this flag. The version of the CAPLIB^2 APL interpreter that has memory debugging turned on in these ten routines is **apldeb2**, created in Linux with this shell script, called **getdeb2**:

getdeb2

```
# cap2/bat/getdeb2
cd ~/cap2/source
cp ~/cap2/include/apldebug.h1 ~/cap2/include/apldebug.h
trap "cp ~/cap2/include/apldebug.h0 \
    ~/cap2/include/apldebug.h" EXIT
buglib2
```

```
 7 linkdeb2
 8 exit
```

Line 2 in **getdeb2** simply changes the current working directory to that containing the CAPLIB^2 source. Line 3 sets **apldebug.h** such that it contains **"#define APL_DEBUG 1."** Lines 4 and 5 set a trap so that any exit from the script causes **apldebug.h** to be set back to **"#define APL_DEBUG 0,"** preventing the inadvertent compilation of CAPLIB^2 routines with memory debugging turned on. Lines 6 and 7 successively invoke shell scripts **buglib2** and **linkdeb2**, both shown below.

buglib2

Shell script **buglib2** builds library **cap2/lib/libbug2.a** by first copying libcap2.a to it, then recompiling all of the routines related to memory debugging:

```
 1 # cap2\bat\buglib2
 2 # Recompiles all of the source that uses mem debugging.
 3 # apldebug.h should be set to 1 or 0 according to whether
 4 # debugging is done.
 5 echo "Creating $HOME/cap2/lib/libbug2.a"
 6 echo "First I'll copy libcap2.a to libbug2.a."
 7 cp $HOME/cap2/lib/libcap2.a $HOME/cap2/lib/libbug2.a
 8 cd $HOME/cap2/source
 9 echo "Now I'll recompile stuff with mem debugging on."
10 for i in aplfree aplinit aplmall errinit errstop \
11          execexed funcexec funcexed funcexee funcsusp
12 do
13   echo "Doing $i"
14   zad $i
15 done
16 exit 0
```

After a few comments and **echo**s to the terminal telling what's going to happen, **buglib2** gets down to business in line 7 by replacing **libbug2.a** with a fresh copy of **libcap2.a**, the complete archive of CAPLIB^2 subroutines. Then lines 10 thru 15 define a loop in which **zad** is invoked for each of the ten CAPLIB^2 subroutines listed on lines 10 and 11, and which compile additional memory debugging statements if APL_DEBUG is 1.

zad

```
# cap2/bat/zad
```

```
2 zd $1 $2 $3 $4
3 rc=$?
4 case $rc in
5  0) bug2 $1;;
6  *) echo "bad compile = $rc"
7      read ltr ;;
8 esac
```

zd

The **zad** shell script calls **zd** in line 2 to compile the specified source, and if the return code indicates no errors, invokes shell script **bug2** to add the object file to archive **libbug2.a**:

```
# cap2/bat/zd, called by zad.
cd ~/cap2/source
cp ~/cap2/include/apldebug.h1 ~/cap2/include/apldebug.h
trap "cp ~/cap2/include/apldebug.h0 \
     ~/cap2/include/apldebug.h" EXIT
z $1 -g $2 $3 $4
exit
```

Script **zd** ensures that **apldebug.h** sets APL_DEBUG to 1, and calls **z** to compile the source. Before **zd** exits, it ensures **apldebug.h** sets APL_DEBUG to 0, so programs compiled and archived in **libcap2.a** won't inadvertently include debugging code.

z

```
# cap2/bat/z
gcc -c -I$HOME/cap2/include \
    -I$HOME/include $2 $3 $4 $5 $6 $1.c
```

bug2

```
# cap/bat/bug2, called by zad.
echo "Linking $1 to $HOME/cap2/lib/libbug2.a now."
ar rs $HOME/cap2/lib/libbug2.a $1.o
rm $1.o
```

linkdeb2

```
# cap2/bat/linkdeb2, called by getdeb2
```

```
echo "Linking apldeb2, from cap2/lib/libbug2.a."
cd ~/cap2/source
zd apldebug
ld -g -o $HOME/bin/apldeb2 -dll-verbose \
     -m486 -non_shared /usr/lib/crt0.o \
     apldebug.o \
     $HOME/src/misc/mem.o \
     $HOME/cap2/lib/libbug2.a \
     -L/usr/lib/gcc-lib/i486-linux/2.6.3 \
     -lcurses -ltermcap -lgcc -lg -lgcc -lm
rm apldebug.o
exit
```

Shell script **linkdeb2** compiles **apldebug.c**, a special version of **apl.c**, which is the main routine of the CAPLIB^2 APL interpreter. Code generated for memory debugging is very slow, and uses lots of processor time. Having a separate source for debugging, rather than a single source with C preprocessor directives, helps to ensure debugging code doesn't get mixed in with the real stuff.

apldebug.c

Apldebug.c declares three routines contained in the mem.c source program written by Walter Bright -

```
#if APL_DEBUG
     void mem_check(void);
     void mem_init(void);
     void mem_term(void);
#endif
```

and generates this call when processing is concluded:

```
#if APL_DEBUG
     mem_term(); /* terminate the memory debugging package */
#endif
```

The **mem_init** routine is called in CAPLIB^2 subroutine **aplinit**:

```
#if APL_DEBUG
void mem_init();
mem_init(); /* initialize the memory debugging package */
#endif
```

aplmall

Each time CAPLIB[2] routine **aplmall** is called to allocate memory, or **aplfree** to free it, a call to **mem_check** is made, if it was compiled in by having APL_DEBUG set to 1. **Mem_check** checks special values inserted before and after allocated memory to detect pointer underruns and overruns, and ensures that freed pointers were previously allocated, and haven't already been freed.
Aplmall has this code in it to divert calls from **malloc** to **mem_malloc**:

```
#if APL_DEBUG
#define malloc mem_malloc
#endif
```

aplfree

Aplfree has similar code to divert calls from **free** to **mem_free**:

```
#if APL_DEBUG
#define free mem_free
#endif
```

Appendix I

CAPLIB2 Module List

This appendix contains a complete list of the modules in **CAP.LIB**. Many of the names listed here you won't see documented elsewhere in this book because they're for internal use, and aren't meant for direct invocation by your C program. The TEXinput for the table below was created under Linux with these commands:

```
ls cap2/source | cut -d. -f1 | sort > ls.out
apl2 < work > work.out
```

Work.out contained these CAPLIB2 statements:

```
)read ls.out
items#4Xrows#S(1YRfile)%cols#4
suffix#itemsR(3R‘Z’ &’),‘Z’ \\’
(rows,colsX2)R(itemsY‘Z[2]file),[1.5]suffix
)off
```

acosh	acoshx	acosx	allcopy
and	andp	apl	aplclean
aplclear	aplcomex	aplcopy	aplcpy
aplcpya	aplcpyb	aplcpyc	aplcpyd
aplcpywr	apldebug	apldrop	apledcl
apledcm	apleddl	apledfm	apledfr
apledfx	apledit	aplediu	aplediv
aplediw	aplediz	aplediy	aplediz
apledmd	apledme	apledmf	aplednn
apledno	aplednw	apledpf	apledpq
apledpr	apledps	apledqu	apledrc
aplerase	aplfill	aplfont	aplfont
aplfont	aplfopn	aplfree	aplgetc
aplgetch	aplgroup	aplgrouq	aplgrour
aplgrous	aplgrp	aplgrps	aplgrpsb
aplinit	apllib	aplload	aplloae
aplloaf	apllov1	aplmall	aplmatch
aplmatci	aplmatcj	aplmatck	aplmsg
aplname	aplnamsb	aplnest	aplnewl
aplparfn	aplparse	aplparsf	aplparsg
aplpnam	aplpnum	aplprint	aplputch
aplquad	aplquae	aplquote	aplread
aplreae	aplrun	aplsave	aplsavf
aplscan	aplsi	aplsrand	aplstr
apltype	apltypf	aplwksp	aplwksq
aplwksr	aplwkst	aplwrite	aplwsid
asinh	asinhx	asinx	assign
atanh	atanhx	atanx	avladd
avladdsb	avlsrch	avltree	axesok
axisdft	axispre	axistest	binom
binomp	binomx	bug	bug
bug	cat	catlamcm	cdyadic
ceil	ceilp	ceilx	ceq
ceqp	ceqpd	charcode	chrcopy
chrvect	circulap	circular	circulas
circulat	circulav	circulaw	circulax
circulay	circulaz	cne	cnep
cnepd	codechar	comexpa	comexpb

complex	compress	comprest	compute
conform	conjug	conjugp	conjugx
convect	convert	cplxin	creduce
creduces	cscan	dabs	dabsp
dabsx	dbland	dblandp	dblcopy
dbleq	dbleqp	dblge	dblgep
dblgt	dblgtp	dblle	dbllep
dbllt	dblltp	dblmax	dblmaxp
dblmin	dblminp	dblnand	dblnandp
dblne	dblnep	dblnor	dblnorp
dblnot	dblnotp	dblor	dblorp
deal	decodbl	decode	depth
depthsub	dequeue	derived	dervfree
digits	disclose	disclosf	disclosg
disclosh	divide	dividep	dividex
dottran	drop	drop2	dtacopy
dublin	dublins	dyadcom	dyadic
dyadicm	dyadicp	dyadopec	dyadoper
dyadrun	dyadrup	dyadset	each
eachalc	eachdyad	eachdyae	eachdyaf
eachdyag	eachdyah	eachmona	eachmonb
eachtran	eachwrk	edline	edlinf
edling	enclose	enclosf	encodbl
encode	encodeb	encodec	endoper
enlist	enlista	enlistb	enqueue
eq	eqp	eqx	errinit
errstop	execalt	execasgn	execaxis
execaxit	execaxiv	execbrkt	execdot
execdota	execdotc	execdotd	execdote
execdotf	execdotg	execdyac	execdyad
execdyah	execdyaj	execdyak	execdyal
execdyam	execdyan	execdyap	execdyaq
execdyar	execdyas	execdyat	execdyau
execdyav	execexec	execexed	execexee
execexef	execexeg	execexeh	execexei
execexej	execexek	execfree	execfun
execgeto	execgetp	execgetq	execgetr
execgets	execglc	execindx	execindy

execinit	execjot	execjot2	execjota
execjotc	execjotd	execmona	execmong
execmonh	execmoni	execmonj	execmonk
execmonq	execmons	execmsg	execnexs
execnext	execnexu	execnila	execopck
execoptk	execout	execpars	execpop
execqcr	execqdl	execqex	execqfx
execqfxa	execqfxb	execqfxc	execqfxd
execqfxe	execqfxf	execqfxg	execqfxh
execqfxi	execqfxj	execqfxk	execqfxl
execqfxm	execqnc	execqncs	execqnl
execqnls	execqnlt	execqnlu	execqts
execqtz	execqtzs	execspec	execsped
execspee	execspef	execspeg	execspeh
execspei	execspej	execspek	execspel
execspen	execspep	execspeq	execsper
execspes	execspet	execspeu	execspex
execspey	execspez	execterm	exectok
execute	executf	executg	expand
expane	expanf	expp	expunge
expungf	expx	factorl	factorlp
factorlx	fifo	find	finda
findb	first	floorp	floorx
font8x16	form	formata	formatb
formatc	formatd	formate	formatf
formatg	formath	formati	formatj
formatk	formatl	formatm	formato
formatp	formatq	formatr	formats
formatt	formatu	formatv	formatw
formatx	formaty	formatz	formben
formbfn	formbfv	formbfw	formbfx
formbtk	formdfs	formdft	formdfu
formdfv	formdfw	formdfx	formdfy
formdfz	formldm	formota	formotb
formsci	formscj	formspp	funcexec
funcexed	funcexee	funcexef	funcgoto
funcinit	funcmain	funcmsg	funcopy
funcstop	funcsusp	funcsusq	functrac

functrgo	functrst	funinit	gammax
ge	genint	gep	getcb
getcbi	getch	getchrv	getcode
getfact	gettcom	gettext	getvect
grade	grade2	grade2a	gradedn
gradedn2	gradesub	gradeup	gradeup2
gt	gtp	iabs	iabsp
iand	iandp	ibinom	ibinomp
iconjug	iconjugp	idivide	idyadic
ieq	ifactorl	ifactorp	ige
igep	igt	igtp	ile
ilep	ilt	iltp	imax
imaxp	imin	iminp	iminus
imod	imonadic	inand	inandp
indexgen	indexm	indexma	indexmb
indexno	indexof	indexv	indices
indxsub	ine	inegatip	inegativ
innrcom	innrprd	innrprda	innrprdb
innrprdc	innrprdd	innrprdp	innrprdx
inor	inorp	inot	intcopy
integer	integes	intone	inttran
intzero	invert	ior	iorder
iorp	iplus	ireduce	ireduces
iresidu	iroll	irollp	iscalar
iscalar	iscan	isign	isignp
itimes	ivalue	laminate	le
leafdel	lep	lifo	litvect
logamma	logp	logrithm	logrithp
logrithx	logx	lovfscb	lovfsfn
lovfsfo	lovfsfp	lt	ltp
match	matchget	matchok	matchoks
matchokt	matchsub	matinv	matmult
mdivarg	mdivide	member	memixb
memixc	memixcm	memixd	memixe
memixf	minus	minusp	minusx
mod	monadic	monfront	multset
namelen	nand	nandp	ne
negative	negativp	negativx	nep

nestchar	newderiv	newtok	nex
nomatchp	nor	norp	not
nreduce	nreduces	nscan	nwise
nwisea	nwiseb	nwisec	nwised
offtptr	or	orp	outrprd
outrprdb	outrprdc	outrprdd	outrprdp
outrprdx	partitn	partito	partitp
perm	pick	pickdyad	pickit
picks	pitimes	pitimesp	pitimesx
places	plus	plusp	plusx
pop	popnesc	popnest	popnesu
popnesv	popnesw	popnesx	power
powerp	powerpx	powerx	precisn
preduce	preduces	preducet	ptrtoff
qquadin	qquadout	quadin	quadinit
quadname	quadnami	quadout	quadoutc
quadrec	quoteit	ravel	ravel2
ravel2a	ravelck	real	recip
recipp	recipx	redscan	reduce
reducecm	reducef	reducesb	regress
regrest	relative	reshape	residue
residuep	residuex	reverse	revrot
rmonadic	roll	rollp	rotate
rotatesb	savfscb	savfsfn	savfsfo
savfsub	scalar	scalax	scalay
scalaz	scan	scanfrnt	setix
shape	signdbl	signdblp	signdblx
signum	slasherr	slashtra	slashtrb
slashtrc	squadix	strread	strwrite
take	take2	takeck	takefill
takeit	takepset	takesub	temp
times	timesp	timesx	tokcopy
transpos	transpot	transpou	treenode
treeroot	treesrch	typeget	value
vectin	vector	vectors	without
xeq	xeqp		

Appendix J

LOCAL AND GLOBAL NAMES

It's difficult in a reference manual to decide when you've said enough about a particular topic. The description of local and global names in the main section of this book is complete, yet I felt more detail might be useful to some. If you're familiar with PL/1 and its nested scoping rules, you'll have little trouble understanding the difference between local and global variables in APL. It'll be more difficult if you've only been exposed to the more limited scoping rules of C.

Consider the following APL statements, with explanations in italics interspersed:

```
apl> Lcr 'demo'

    a # b demo c
    a # b + c # c + d # b
```

Lcr *is documented in detail on page 62, but briefly, it produces a character matrix containing a representation of the specified defined function.*

```
apl> (a#23),(b#45),(c#67),d#89

23 45 67 89
```

Variables **a, b, c,** *and* **d** *are respectively set to* 23, 45, 67, *and* 89. *These variables are all considered* **global** *to defined function* **demo**.

```
apl> 1 + c demo d

224

apl> a,b,c,d

23 45 67 67
```

Defined function **demo** *is passed control. But first, local variables* **b** *and* **c** *are set respectively to* 67 *and* 89, *the present values of global variables* **c** *and* **d**. *The name* **a** *is also localized so that any reference to it inside* **demo** *won't access or affect the value of the global instance. Next, the statement inside* **demo** *is executed:*

```
apl> a # b + c # c + d # b
```

This expression is evaluated right-to-left, as of course all APL expressions are, in the following steps:

- *Global variable* **d** *is assigned the value* 67, *the present value of local variable* **b**.
- *Local variables* **b** *and* **c** *are added,* $67 + 156$, *yielding* 223, *which is assigned to local variable* **a**.
- *Defined function* **demo** *finishes. The present value of local variable* **a**, 223, *is returned to the calling expression, after freeing storage for local variables* **a, b,** *and* **c**.
- *Finally,* 1 *and* 223 *are added and displayed. Notice the result of the APL statement* **a,b,c,d** *indicates that only the value of variable* **d** *was altered in* **demo**.

The terms **local** *and* **global** *are relative, and a given instance of a variable can be both* **local** *to the present defined function, and* **global** *to another defined function called by the first. For example, suppose we add a new defined function called* **also** *and add a call to it in demo:*

```
apl> Lcr 'demo'

    a # b demo c
    a # b + c # c + d # also b

apl> Lcr 'also'

    a # also e
    a # e + d + b # 32

apl> a,b,c,d#89

23 45 67 89

apl> c demo d

309

apl> a,b,c,d

23 45 67 188
```

Demo *is passed control after setting local* **b** *and* **c** *respectively to* 67 *and* 89. *Again,* **a** *is localized to protect the global instance. Then the statement in* **demo** *is executed:*

```
apl> a # b + c # c + d # also b
```

Also *is passed control after setting a new localized instance of* **e** *to* 67, *and localizing name* **a** *to protect global instances of it (at this point, there are three instances of* **a**). *Next, the statement in* **also** *is executed:*

```
apl> a # e + d + b # 32
```

32 *is assigned to the instance of* **b** *that is global to* **also** *and local to* **demo**. *Then* 32 *is added to* 89 *from the only instance of* **d**, *global to both* **demo** *and* **also**. *The result,* 121, *is added to* 67 *from the local instance of* **e**, *and* 188 *is assigned to the local instance of* **a**.

Also *terminates, freeing instances of* **a** *and* **e**, *and* 188 *is returned to the calling expression, where it is assigned to global* **d**. 89 *and* 188 *are added yielding* 277, *assigned to local* **c**. 277 *plus* 32 *from* **b** *assigned in* **also** *when it was global to it, yield* 309 *assigned to local* **a**.

Finally **demo** *terminates, its locals are freed, and the result* 309 *is displayed, followed by the catenation of* **a, b, c,** *and* **d**, *showing only* **d** *was altered. Because of the context in which they were used, names* **a, b,** *and* **c** *in* **demo** *were automatically defined to be local to it. Variables like* **d** *could also have been made local to* **demo** *by appending them to the* **function header**, *separated by semicolons, as in* **demo2** *below:*

```
apl> Lcr 'demo2'

    a # b demo2 c;d;e
    a # b + c + d # 2 + e # 3

apl> a,b,c,d#89

23 45 67 89

apl> c demo2 d

161

apl> a,b,c,d

23 45 67 89
```

Appendix K

DEL EDITOR SAMPLE SESSION

Editing a New Function

Here's a sample session that begins by creating a defined function called **sum** (annotations are in italics):

The APL interpreter prompts with "`APL>`*". I tell it I want to enter the del editor for function* **sum***:*

```
apl> Gsum
```

The editor generally prompts for input by displaying the bracketed statement number it expects you to enter next. In this case it has prompted me to enter statement `[1]`*. Because the function is new, the editor assumes I entered the function header when I entered "*`Gsum`*". It thinks* **sum** *is going to be niladic without a returned value. That's not the case, so I tell him I want to re-enter the function header by keying* `[0]`*:*

```
[1]      [0]
```

The editor prompts me with "`[0]`*" and I supply the new function header for sum:*

```
[0]      z#sum x
```

The editor prompts me with the next sequential statement number, and I respond with the statement:

```
[1]       z#+/x
```

The editor prompts me with the next sequential statement number. I ask to see the entire defined function:

```
[2]       [L]

      G z#sum x
[1]     z#+/x
      G
[2]
```

The editor displays the entire function, placing a `G` *in front of the function header and after the last statement. It also lists bracketed statement numbers at the left margin. Finally it prompts me again to enter statement 2, to which I respond "*`G`*" to tell it I'm done editing, and I'm once again prompted with "*`APL>`*":*

```
[2]      G
apl>
```

Inserting a Statement

Suppose I want to add a comment to defined function **sum** telling what it does:

```
apl> Gsum
[2]      [0.1] " provides vector sums
[0.11]   [L]
        G z#sum x
[0.1]    " provides vector sums
[1]       z#+/x
        G
[2]      G
apl>
```

Deleting a Statement

Then I change my mind and decide to delete the comment, displaying the entire function before and after:

```
apl> Gsum[L]
        G z#sum x
[1]      " provides vector sums
[2]       z#+/x
```

```
        G
[3]      [H1]
[3]      [L]
        G z#sum x
[2]      z#+/x
        G
[3]      G
apl>
```

Editing Statements

Now I'll use the statement editing capabilities to transform **sum** into defined function **mean**:

```
apl> Gsum[L]
        G z#sum x
[1]       z#+/x
        G
```

First, here's the total replacement of the function header:

```
[2]      [0]
[0]       z#mean x
[0.1]
```

Now to change statement `[1]` *so it produces the vector means rather than the sums:*

```
[0.1]    [1L0]
[1]      z#+/x
```

Entering "`[1L0]`*" above told the editor to display statement* `[1]` *with the cursor positioned at the end of the statement, but one line lower on the display. With standard APL, now you're expected to position the cursor and key in letters or decimal digits indicating where you want blanks inserted in the statement, and how many. A "/" means delete the character in the statement at that position. You can use this if you want to, it'll work. But I just press* **Enter** *at this point:*

```
[1]      z#+/x(
[1]      z#+/x(
               ^ unbalanced parentheses
[1]      z#(+/x)%-1Y1,Rx
[2]      G
apl>
```

The editor responds by re-displaying the statement with the cursor positioned on the same line at the right end of the statement. I deliberately keyed the left parenthesis to get the editor to issue the "unbalanced parenthesis" message and enter a special edit mode which is not standard APL. In this mode:

- *The left and right arrow cursor keys on the PC keyboard move the cursor to new positions in the statement, without changing anything.*
- *The* **Backspace** *key moves the cursor left, and at the same time deletes a character.*
- *Pressing any of the alphanumeric keys, e.g. "K", inserts that character in the statement at the cursor position, and the cursor is shifted right one position.*

Now let's try the new mean function to see if it works:

```
apl> Gmean[L]G

[0]    G  z#mean x
[1]       z#(+/x)%-1Y1,Rx
       G

apl> mean 1 2 3 4 5

3

apl> mean 5

5

apl> mean 3 4RI12

2.5 6.5 10.5

apl>
```

Appendix L

CapLib^2 Distribution Media

If you get the CapLib^2 distribution from me, it'll be on a 3.5-inch high-density (1.44 Megabyte) diskette, in two files:

- **cap2.tar.gz**: This is the version of CapLib^2 for Linux, created with these commands (see cap2/bat/cap2tar):

```
cd; tar czf /tmp/cap2.tar.gz cap2
```

- **cap2dos.tgz**: This is the version of CapLib^2 for DOS, created with this command (see cap2/bat/cap2tar.dos):

```
cd /dosd/my
tar czf /tmp/cap2dos.tgz \
     cap2/copyrite.txt \
     cap2/file_id.diz \
     cap2/gnu.txt \
     cap2/install.txt \
     cap2/read.me \
     cap2/bin \
     cap2/handbook \
     cap2/help
```

To extract from either tar file, use a command like one of these, after changing to the directory you want **cap2** to be a sub-directory of:

```
tar xzf /mnt/cap2dos.tgz
tar xzf /mnt/cap2.tar.gz
```

A Tour of the CAPLIB^2 Files

Regardless of which tar file you extract from, each will expand to very similar hierarchies of files, the biggest difference being the CAPLIB^2 source is not included in the DOS version. If want the source, get Linux!

- **cap2/INSTALL.TXT**: Describes how to install CAPLIB^2. In the Linux distribution this file is in uppercase so it lists first in a directory list from an **ls** or **ll** command.
- **cap2/READ.ME**: Describes how to use CAPLIB^2. It's a super-condensed version of this book.
- **cap2/copyrite.txt**: A brief declaration that the software is protected under copyright.
- **cap2/file_id.diz**: A five-line summary description of CAPLIB^2.
- **cap2/gnu.txt**: A text file of the standard GNU General Public License, also contained in Appendix Q of this book.
- **cap2/keys.txt**: A text file describing how the keyboard maps to CAPLIB^2 characters. For example, it indicates **R** represents the APL **rho** symbol (ρ).
- **cap2/bat**: This is a subdirectory with shell scripts in Linux, and BAT files in DOS. Only the Linux scripts will be described here:
  - **cap2/bat/bug2**: Called by **zad** (see below) to add an object file to the **cap2/lib/libbug2.a** library.
  - **cap2/bat/buglib2**: Creates **cap2/lib/libbug2.a**, used in debugging. Uses the **zad** script to compile special debugging code in selected CAPLIB^2 routines. See Appendix H on **Memory Debugging**.
  - **cap2/bat/cap2tar**: Creates **cap2.tar.gz**.
  - **cap2/bat/cap2tar.dos**: Creates **cap2dos.tgz**.
  - **cap2/bat/caplib2**: Rebuilds **cap2/lib/libcap2.a**, the CAPLIB^2 subroutine library, from scratch.
  - **cap2/bat/getapl2**: Compiles and links **cap2/source/apl.c**, producing **cap2/bin/apl2**.
  - **cap2/bat/getcap2**: Re-creates all the C source programs in **cap2/source** by copying all the files in **/dosd/my/cap2/source**, removing the $\hat{\text{M}}$ that DOS needs at the end of each line.

- **cap2/bat/getdeb2**: Creates **cap2/bin/apldeb2**, a version of the CAPLIB^2 APL2 interpreter, prepared for debugging.
- **cap2/bat/lib2**: Called by **za**, for example, after compiling a CAPLIB^2 subroutine to add the object to **cap2/lib/libcap2.a**.
- **cap2/bat/linkdeb2**: Called by **cap2/bat/getdeb2** to link, after compiling **cap2/source/apldebug.c**, to produce **cap2/bin/apldeb2**.
- **cap2/bat/validate**: For every file in **cap2/sample**, calls **cap2/bat/validsub** to run the CAPLIB^2 APL2 interpreter against it, redirecting the output to directory **cap2/validate/new**, then compares that file against predicted output in **cap2/validate/old**.
- **cap2/bat/validsub**: Called by **cap2/bat/validate**.
- **cap2/bat/z**: Compiles a C source program using **gcc**, with the proper CAPLIB^2 **include** directories specified.
- **cap2/bat/z**: Calls **cap2/bat/z** to compile a C source program, and if the compile is without error, calls **cap2/bat/lib2** to add the resulting object to **cap2/lib/libcap2.a**.
- **cap2/bat/zad**: First calls **cap2/bat/zd** to compile a C source program with debugging turned on, then calls **cap2/bat/bug2** to add the object to **cap2/lib/libbug2.a** if the compile was without error.
- **cap2/bat/zd**: Compiles a C source program with debug turned on.
- **cap2/bat/zl**: Compiles and links a C source program, using the CAPLIB^2 **include** files and subroutine library.

• **cap2/bin**: In Linux, subdirectory **cap2/bin** contains just one binary, **apl2**. In DOS there are three executables:

- **cap2/bin/apl2.exe**: The CAPLIB^2 APL interpreter.
- **cap2/bin/aplfont.exe**: Sets the DOS font buffer such that APL symbols are displayed. See **cap2/bat/aplf.bat**.
- **cap2/bin/font8x16.exe**: Restored the DOS font buffer to its original state. See **cap2/bat/aplf.bat**.

• **cap2/handbook**: Everything in this subdirectory is related to defined functions modelled after those listed in the "**APL Handbook of Techniques**" published and copyrighted by IBM in April, 1978. These functions were used in the initial testing of the CAPLIB^2 APL functions.

- **cap2/handbook/make**: This subdirectory contains a series of text files that, when processed by the CAPLIB^2 interpreter, yield the workspace in **cap2/handbook/wksp** (see below).
- **cap2/handbook/source**: This subdirectory contains text files, each containing CAPLIB^2 statements that build one of the defined functions in **cap2/handbook/wksp**. For example, file **add.txt** contains the definition for the **add** defined function in **cap2/handbook/wksp/ac.apl** (see below).
- **cap2/handbook/wksp**: This subdirectory contains nine CAPLIB^2 workspaces, **ac, df, gi, jl, mo, pr, su, vx,** and **yz,** all having the suffix **.apl**. Each workspace contains defined functions whose names begin with a letter in the range indicated by the workspace name. For example **ac.apl** contains defined functions **add** through **cvec**.

* **cap2/help**: This directory contains information helpful in using the CAPLIB^2 APL2 interpreter.
  - **cap2/help/display.txt**: IBM distributes a display function that graphically portrays the structure of APL2 variables. This is a text file containing the CAPLIB^2 version. See the tutorial section, on page 14.
  - **cap2/help/help.apl**: This is a CAPLIB^2 workspace that offers help when using the interpreter. It's described in the tutorial section, on page 4.

* **cap2/include**: This subdirectory contains files **#included** in CAPLIB^2 subroutines during compilation. Examples are **aplcb.h** and **apltoken.h**.

* **cap2/sample/c**: Contains the source for 126 sample C programs, most of which are shown in this book. They were all compiled and executed, to ensure their validity.

* **cap2/sample**: Contains 290 text files, each containing CAPLIB^2 APL2 statements. These were all fed to the interpreter, with output redirected to files in subdirectory **cap2/validate/old**. Any time a change is made to the CAPLIB^2 interpreter, all these sample files are then fed to the new interpreter, with the output compared to that in **cap2/validate/old** to ensure it's producing correct answers.

* **cap2/source**: Contains 748 C source files, all of the source for the CAPLIB^2 subroutines. This subdirectory in only included in the Linux version of CAPLIB^2.

Appendix M

Display Function

Listed below is a copy of the DISPLAY function as distributed with IBM's APL2, except it's listed in CAPLIB2 form. A copy of it is distributed in the CAPLIB2 HELP workspace. The following rules apply:

- The output of the DISPLAY function is a series of nested boxes containing the formatted items from the array.
- The top and left borders of each box display symbols that indicate the rank of the array. Lack of a symbol in the top and left borders indicates a scalar.
- symbols in the bottom border indicate either the type of data, or that the array is nested.

Border Symbols of the DISPLAY function	
Symbol	**Meaning**
$	At least a vector
U	Matrix or array
O or @	Empty along this axis
~	Numeric
+	Mixed character and Numeric
no symbol	Character
E	Nested
-	Scalar blank

Table M.1: Symbols used in the borders by the DISPLAY function.

```
d#s display a;Lio;r;c;hl;hc;ht;hb;vl;vb;v;w;n;b;t
" display a general array in pictorial form
"  normal call is monadic.  dyadic call used only in
"    recursion to specify display rank, shape, and depth
Lio#0
C(0=Lnc 's')/'s#Ra'
r#YR,s              " pseudo rank.
c#'++++'            " ur, ul, ll, and lr corners.
hl#'-'              " horizontal line.
hc#hl,'@$',hl,'~+E' " horizontal borders.
ht#hc[(0<r)X1+0<Y-1Y,s]
"w#,0‘=‘1Y0R‘Z(1SRa)Ya
"  this expression will produce 1 if all items
"  in variable a are numeric, or will fail with
"  a domain error otherwise.
"  a      1 if 0 items in a, otherwise count of
"  a      items in a.
"  a      b
"  a      bbbbbb
"  aaaaaaaaaaaaaaaaa
hb#hc[3+3D('2X~a‘=Fa' Lea '1+YE0X(1SX/Ra)Y,a')+3X1<RRs]
vl#'|'                   " vertical line.
vb#vl,'0U'               " vertical border.
v#vb[(1<r)X1+0<-1Y-1U,s]
C(0ERa)/'a#(1SRa)R‘ZYa'  " show prototype of empties
$(1<‘=a)/gen
$(2<RRa)/d3
d#Fa                     " simple arrays
w#1YRd#(-2Y1 1,Rd)Rd
n#-1+1URd
$(0=RRa)/ss
t#c[0],(wRvl),c[3]
d#(c[1],v,((w_1)Rvl),c[2]),((ht,nRhl),[0]d,[0]hb,nRhl),t
$0
ss:hb#((0 ' ')=Y0R‘Za)/' -'
d#' ',(' ',[0]d,[0]hb,nR' '),' '
$0
gen:d#Fdisplay‘1a        " enclosed ...
n#dV.^=' '
d#(nV~1‘0n)‘/d
d#(V‘/~'  '‘Ed)/d
d#((1,Rs)Rs)display d
```

```
$(2ZR,s)Ud3e,0
d3:d#0 -1U0 1UF‘Za       " multi-dimensional
w#1YRd
n#-1+1URd
t#c[0],(wRvl),c[3]
d#(c[1],v,((w_1)Rvl),c[2]),((ht,nRhl),[0]d,[0]hb,nRhl),t
d3e:n#-2+R,s
v#c[nR1],[0]vb[1+0<-2U,s],[0](((-3+YRd),n)Rvl),[0]c[nR2]
d#v,d
" copied from IBM’s tryapl2 workspace
```

Appendix N

Example: Inventory

IBM's APL Language Reference lists a good example on pages 3 and 4 illustrating many of the fundamentals of the language, including the enhancements of APL2. This appendix contains their illustration, paraphrased in CAPLIB2 symbols.

<table>
<tr><th colspan="2">Expressions for Maintaining and Reporting Inventory Quantities, Part 1</th></tr>
<tr><th>CAPLIB² Expression</th><th>Comment</th></tr>
<tr><td>For each group (men, women, children), enter the number of shoes of different styles in stock:
<pre>
men#45 75 15
women#35 75 15 45 95
children#35 0 55 15
</pre></td><td>Specification of variables (page 47) and use of arrays (page 8).</td></tr>
<tr><td>Determine the total number of men's shoes:
<pre>
+/men
135
</pre></td><td>Slash (/) operator (page 275) with add (+, page 258) as operand.</td></tr>
<tr><td>Determine totals for each group:
<pre>
+/‘1 men women children
135 265 105
</pre></td><td>Vector notation (pages 21 and 81) and each (‘) operator (page 103) with derived function (page 16) summation (+/) as operand.</td></tr>
<tr><td>Determine the total number of shoes in stock:
<pre>
+/E men women children
505
</pre></td><td>Enlist (E, page 116).</td></tr>
<tr><td>Represent stock as a single variable
<pre>
stock # men women children
stock
 45 75 15 35 75 15 45 95 35 0 55 15
+/‘1 stock
135 265 105
+/E stock
505
</pre></td><td>Nested array (page 14).</td></tr>
<tr><td>Display the inventory information as a table:
<pre>
‘X[1]stock
45 35 35
75 75 0
15 15 55
 0 45 15
 0 95 0
</pre></td><td>Disclose with axis (‘X[1], page 89), which fills with zeros where data was not provided.</td></tr>
</table>

<table>
<tr><th colspan="2">Expressions for Maintaining and Reporting Inventory Quantities, Part 2</th></tr>
<tr><th>CAPLIB2 Expression</th><th>Comment</th></tr>
<tr><td>Describe what the numbers represent:

styles#’oxfords’ ’loafers’ ’sneakers’

styles#styles,’sandals’ ’pumps’

groups#’mens’ ’womens’ ’childrens’</td><td>Catenate (<code>,</code> page 63), and character data (page 81).</td></tr>
<tr><td>Add row and column headings to the table:

(’ ’,groups),[1]styles,‘X[1]stock

mens womens childrens
oxfords 45 35 35
loafers 75 75 0
sneakers 15 15 55
sandals 0 45 15
pumps 0 95 0</td><td>Catenate with axis (page 63), and intermixed character and numeric data (page 81).</td></tr>
<tr><td>Maintain costs for each style in each group:

costs#(39 19 29) (35 15 29 18 45)

costs#costs,‘Z25 16 21 12.5

costs

39 19 29 35 15 29 18 45 25 16 21 12.5</td><td>Parenthesis (page 14) and Enclose (‘Z, page 106)</td></tr>
<tr><td>What is the cost of men’s loafers?

1 2‘Xcosts

19</td><td>Pick (‘X, page 252).</td></tr>
<tr><td>Change the cost on men’s loafers:

(1 2 ‘X costs) # 20

costs

39 20 29 35 15 29 18 45 25 16 21 12.5</td><td>Selective assignment, or specification (#, page 48).</td></tr>
</table>

Expressions for Maintaining and Reporting Inventory Quantities, Part 3	
CAPLIB2 Expression	**Comment**
Determine retail costs if markups from wholesale costs for men's, women's, and children's shoes are 60, 70, and 80 percent, respectively. `prices # costs X 1 + .6 .7 .8` `,[IO] prices` `            62.4 32 46.4` ` 59.5 25.5 49.3 30.6 76.5` `       45 28.8 37.8 22.5`	Multiply (`X`, page 326), scalar functions (Appendix D, on page 359), and ravel with axis (`,[]`, page 267).
Identify the stock investment for each group: `groups, ‘1 costs +.X ‘1 stock` `  mens 3690    womens 7870    childrens 2217.5`	Array product (`.`, page 193).
Determine the resulting net profit (total sales minus total cost) for each line in stock: `net#stockXprices_costs` `,[IO]net` `                1053 900 261` ` 857.5 787.5 304.5 567 2992.5` `               700 0 924 150`	Subtract (`_`, page 228).
Determine the net profit by group and the total net profit: `+/‘1net` ` 2214 5509 1774` `+/+/‘1net` ` 9497`	
Identify the group and style that has the largest net profit: `groupstyles#,(groups,‘1’ ’)J.,styles` `(,‘Xnet=S/Enet)/groupstyles` ` womens pumps`	Outer Product (`J.m` page 242), ravel (`comma` (`,`), page 267), maximum (`S`, page 224), equal (`=` , page 119), and disclose (`‘X`, page 89).

Appendix O

Trigonometry

Introduction

Not being a mathematician, I took care to ensure CAPLIB2 gives correct results. When checking my answers, I often found other programs were incorrect, especially pertaining to the inverse functions. I hope I've got it right.

Sine

Figure P.1 on page 436 shows the graph of the trigonometric sine for real numbers, and Figure P.5 on page 440 shows two graphs of the real coefficient (9O1Ox) of the sine for complex numbers, and Figure P.6 on page 441 shows similar graphs of the imaginary coefficient (11O1Ox). Each coefficient cycles in the same manner respectively as the sine and cosine for real numbers, differing only in amplitude. This behavior is consistent with the following equation defining the **sin** function for complex numbers (assume $x = COMPLEX(a, b)$ produces the complex number $a + bi$):

$$SIN(x) = COMPLEX(SIN(a) * COSH(b), COS(a) * SINH(b)) \quad (O.1)$$

Here's an equivalent user-defined **sin** function:

```
apl>Gz#sin x;a;b
[1]       z#((1Oa)X6Ob)+0j1X(2Oa#9Ox)X5Ob#11Ox
[2]       G
apl>(L#1Ox) = sin x#0j1+1 2 3
```

```
1.298458j.634964  1.403119j-0.48905626 .21775955j-1.16344

1 1 1
```

Cosine

Figure P.2 on page 437 shows the graph of the trigonometric cosine for real numbers.

Figure P.7 on page 442 shows two graphs of the real coefficient (9O2Ok) of the cosine for complex numbers. It cycles in the same manner as the cosine for real numbers, differing only in amplitude, consistent mathematical expression defining **cos** with the results of this user-defined **cos** function below ($x = COMPLEX(a, b)$ produces the complex number $a + bi$. This expression defines the cos function):

$$COS(x) = COMPLEX(COS(a) * COSH(b), -SIN(a) * SINH(b)) \tag{O.2}$$

Figure P.8 on page 443 shows four graphs of the imaginary coefficient (11O2Ok) of the cosine for complex numbers. It cycles in the same manner as the sine for real numbers, differing only in amplitude and sign, consistent with equation O.2 above, and with the results of this user-defined **cos** function:

```
apl>Gz#cos x;a;b
[1]       z#((2Oa)X6Ob)+0j1X(_1Oa#9Ox)X5Ob#11Ox
[2]       G
apl>(L#2Ox) = cos x#0j1+1 2 3

.83373j-0.988898  -0.642148j-1.068607 -1.527638j-0.1658444

1 1 1
```

Tangent

```
apl>Gz#tan x;a;b;c
[1]       z#((1Oa)%c)+0j1X(5Ob)%c#(2Oa#2X9Ox)+6Ob#2X11Ox
[2]       G
apl>(L#3Ox) = tan x#0j1+1 2 3

.27175259j1.083923  -0.2434582j1.166736 -0.05916854j.768018

1 1 1
```

Assume that $x = COMPLEX(a, b)$ produces the complex number $a+bi$, $REAL(x)$ produces the original value a, and $IMAG(x)$ the original b, the following mathematical expressions are equivalent to the above CAPLIB2 statement for the *tan* user-defined function:

$$REAL(TAN(x)) = \frac{SIN(2a)}{(COS(2a) + COSH(2b))} \tag{O.3}$$

$$IMAG(TAN(x)) = \frac{SINH(2b)}{(COS(2a) + COSH(2b))} \tag{O.4}$$

Figure P.3 on page 438 shows the graph of the trigonometric tangent for real numbers (3Ok), and Figure P.9 on page 444 shows a graph of the real coefficient (9O3Ok) of the tangent for complex numbers. The graph of 9O3Ok is similar to the one for the tangent for real numbers, differing significantly in that it's continuous at any $\frac{k\Pi}{2}$, k an integer.

Graphing 11O3Ok is more complex, as shown in Figure P.10 on page 445. As —11Ok increases, $11O3Ok$ approaches 1 for positive $11Ok$, and -1 for negative $11Ok$.

Hyperbolic Sine (sinh) for Real Numbers

Figure P.4 on page 439 shows graphs including hyberbolic sine **(sinh)** for real numbers. The mathematical expression used in implementing the CAPLIB2 sinh function **(5Ox)** for real numbers is -

$$sinh(x) = \frac{1}{2}(e^x - e^{-x}) \tag{O.5}$$

Here's a user-defined **sinh** function equivalent to the method used in CAPLIB2 for **sinh** of real numbers:

```
apl>Lfx 'z#sinh x;t' 'z#.5Xt_1%t#*x'

sinh

apl>&/(5Ox) = sinh x#-5+I10

1

apl>(5Ox),[1.5]sinh x

-27.289917 -27.289917
-10.017875 -10.017875
 -3.626860  -3.626860
```

```
 -1.175201  -1.175201
  0.000000   0.000000
  1.175201   1.175201
  3.626860   3.626860
 10.017875  10.017875
 27.289917  27.289917
 74.203211  74.203211
```

Hyperbolic Sine (sinh) for Complex Numbers

Assume $x = COMPLEX(a, b)$ produces the complex number $a+bi$, the following mathematical expression is equivalent to the CAPLIB^2 **sinh (5Ox)** function:

$$SINH(x) = COMPLEX(SINH(a) * COS(b), COSH(a) * SIN(b)) \tag{O.6}$$

Here's a user-defined **sinh** function that illustrates an equivalent implementation to that used in CAPLIB^2 for **sinh** of complex numbers:

```
apl>Gz#sinh x;a;b;c
[1]       z#((5Oa)X2Ob)+0j1X(6Oa#9Ox)X1Ob#11Ox
[2]       G
apl>(L#5Ox) = sinh x#0j1+1 2 3

.634964j1.298458  1.959601j3.165779 5.412681j8.471645

1 1 1
```

Figure P.11 on page 446 shows graphs of the CAPLIB^2 expression **9O5Ox+nX0j1** for various constant values of **n**. This is a family of functions, all with points of inflection at $(0, 0)$, domains of $(-\infty, \infty)$, and with signs and slopes determined by $COS(b)$.

Figure P.12 on page 447 shows graphs of the CAPLIB^2 expression **11O5Ox+nX0j1** for various constant values of **n**. This is a family of functions, all with minima at $x = 0$, and domains of $(-\infty, \infty)$. In other words, they all behave like **COSH** times a constant, as you'd expect from equation O.6 on page 414.

Figure P.13 on page 448 shows the graph of **9O5On+xX0j1**, n a constant, and **x,** the imaginary coefficient, is varied. Except for the amplitude, the graph is like the graph of **cosine**, which is consistent with equation O.6 on page 414.

Figure P.14 on page 449 shows the graph of **11O5On+xX0j1**, n a constant, and **x,** the imaginary coefficient, is varied. As you'd expect from equation O.6 on page 414, the shape of this curve is like that of the **sine(x)**, **x** being real numbers.

Hyperbolic Cosine (cosh) for Real Numbers

Figure P.4 on page 439 shows graphs including hyberbolic cosine **(cosh)** for real numbers. The mathematical expression used in implementing the CAPLIB2 cosh function **(6Ox)** for real numbers is -

$$cosh(x) = \frac{1}{2}(e^x + e^{-x}) \tag{O.7}$$

Here's a user-defined **cosh** function equivalent to the method used in CAPLIB2 for **coshh** of real numbers:

```
apl>Lfx 'z#cosh x;t' 'z#.5Xt+1%t#*x'

cosh

apl>&/(6Ox) = cosh x#-5+I10

1

apl>(6Ox),[1.5]cosh x

27.308233 27.308233
10.067662 10.067662
 3.762196  3.762196
 1.543081  1.543081
 1.000000  1.000000
 1.543081  1.543081
 3.762196  3.762196
10.067662 10.067662
27.308233 27.308233
74.209949 74.209949
```

Hyperbolic Cosine (cosh) for Complex Numbers

Assume that $x = COMPLEX(a, b)$ produces the complex number $a + bi$, the following mathematical expression is equivalent to the CAPLIB2 **cosh (6Ox)** function:

$$COSH(x) = COMPLEX(COSH(a) * COS(b), SINH(a) * SIN(b)) \tag{O.8}$$

Here's a user-defined **cosh** function equivalent to the method used in CAPLIB2 for **cosh** of complex numbers:

```
apl>Gz#cosh x;a;b;c
[1]       z#((6Oa)X2Ob)+0j1X(5Oa#9Ox)X1Ob#11Ox
[2]       G
apl>(L#6Ox) = cosh x#0j1+1 2 3

.83373j.988898   2.032723j3.051898 5.439581j8.429751

1 1 1
```

Figure P.15 on page 450 shows graphs of the CAPLIB2 expression **9O6Ox+nX0j1** for various constant values of **n**. As you'd expect from equation O.8 page 415, all the curves resemble that of constant **k** times **COSH(x)**.

Figure P.16 on page 451 shows graphs of the CAPLIB2 expression **11O6Ox+nX0j1** for various constant values of **n**. Once again, according to equation O.8, they all resemble the graph of constant **k** times **SINH(x)**.

Figure P.17 on page 452 shows the graph of **9O6On+xX0j1**, with n a constant, and **x,** the imaginary coefficient, varied. Except for the amplitude, the graph is like the graph of **cosine**, which is consistent with equation O.8 on page 415.

Figure P.18 on page 453 shows the graph of **11O6On+xX0j1**, n a constant, and **x,** the imaginary coefficient, is varied. As you'd expect from equation O.8 on page 415, the shape of this curve is like that of the **sine** for real numbers.

Hyperbolic Tangent (tanh) for Real Numbers

Figure P.19 on page 454 graphs the hyperbolic tangent **(tanh, 7Ox)** for real numbers. Here's the mathematical expression used in implementing the CAPLIB2 tanh function for real numbers:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{sinhx}{coshx} \tag{O.9}$$

Here's two user-defined functions for tanh:

```
apl>Lfx 'z#tanha x;s;t' 'z#(t_s)%t+s#1%t#*x'

tanha

apl>&/(7Ox) = tanha x#-5+I10
```

```
1

apl>Lfx 'z#tanhb x' 'z#(5Ox)%6Ox'

tanhb

apl>&/(7Ox) = tanhb x

1

apl>(7Ox),[1.5]tanhb x

-0.999329 -0.999329
-0.995055 -0.995055
-0.964028 -0.964028
-0.761594 -0.761594
 0.000000  0.000000
 0.761594  0.761594
 0.964028  0.964028
 0.995055  0.995055
 0.999329  0.999329
 0.999909  0.999909
```

Hyperbolic Tangent (tanh) for Complex Numbers

Equation O.9 on page 416 needs only to be extended slightly to encompass the hyperbolic tangent for complex numbers:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{sinhx}{coshx} = \frac{sinh(a)cosh(b) + cosh(a)sin(b)i}{cosh(a)cos(b) + sinh(a)sin(b)i} \quad (O.10)$$

Here's two user-defined **tanh** functions equivalent to the methods used in CAPLIB2:

```
apl>
Lfx 'z#tanha x;s;t' 'z#(t_s)%t+s#1%t#*x'

tanha

apl>&/(7Ox) = tanha x#-5+I10

1
```

```
apl>Lfx 'z#tanhb x' 'z#(5Ox)%6Ox'

tanhb

apl>&/(7Ox) = tanhb x

1

apl>
(7Ox),[1.5]tanhb x

-0.999329 -0.999329
-0.995055 -0.995055
-0.964028 -0.964028
-0.761594 -0.761594
 0.000000  0.000000
 0.761594  0.761594
 0.964028  0.964028
 0.995055  0.995055
 0.999329  0.999329
 0.999909  0.999909
```

Figure P.20 on page 455 shows graphs of the CAPLIB2 expression **9O7Ox+nX0j1** for various constant values of **n**, and Figure P.21 on page 456 shows graphs of the CAPLIB2 expression **11O7Ox+nX0j1** for various constant values of **n**. For larger magnitudes of **x** the graph rapidly settles toward 1 and -1, consistent with -

$$\lim_{x \to \infty} \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 \tag{O.11}$$

$$\lim_{x \to -\infty} \frac{e^x - e^{-x}}{e^x + e^{-x}} = -1 \tag{O.12}$$

Figure P.22 on page 457 shows the graph of **9O7On+xX0j1**, with n a constant, and **x,** the imaginary coefficient, varied. There are maxima or minima at **9O7On+0j1X(O.5)XsX-1+2XIk** and **9O7On+0j1XOsXIk** for all integer **k**, and for $s = \pm 1$. This is consistent with equation O.10 on page 417, since if $\cos(b) = 0$, it follows that $\sin(b) = 1$ and tanh devolves to $\sinh(a)/\cosh(a)$, in this case a constant. A similar situation arises if $\sin(b) = 0$.

Figure P.23 on page 458 shows the graph of **11O7On+xX0j1**, n a constant, and **x,** the imaginary coefficient, varied. It has roots at $x = n \cdot \pi/2$, maxima at $x = n \cdot \pi + \pi/4$, and minima at $x = n \cdot \pi - \pi/4$, n any integer.

If $x = a + bi$, then -

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{(sinh(a)cos(b) + cosh(a)sin(b)i)}{cosh(a)cos(b) + sinh(a)sin(b)i} \tag{O.13}$$

$$= \frac{(sinh(a)cos(b) + cosh(a)sin(b)i)(cosh(a)cos(b) - sinh(a)sin(b)i)}{cosh^2(a)cos^2(b) + sinh^2(a)sin^2(b)}$$

$$= \frac{\left\{ \begin{array}{l} sinh(a)cosh(a)cos^2(b) - sinh^2(a)cos(b)sin(b)i \\ +cosh^2(a)sin(b)cos(b)i - cosh(a)sinh(a)sin^2(b)i^2 \end{array} \right\}}{cosh^2(a)cos^2(b) + sinh^2(a)sin^2(b)}$$

$$= \frac{sinh(a)cosh(a)cos^2(b) + cosh(a)sinh(a)sin^2(b)}{cosh^2(a)cos^2(b) + sinh^2(a)sin^2(b)}$$

$$+ \frac{cos(b)sin(b)(cosh^2(a) - sinh^2(a))}{cosh^2(a)cos^2(b) + sinh^2(a)sin^2(b)} i$$

Holding **a** a constant devolves the expression for the imaginary coefficient above into -

$$\frac{cos(b)sin(b)(k-m)}{k \cdot cos^2(b) + m \cdot sin^2(b)} = \left(\frac{f}{g}\right)(b) \tag{O.14}$$

Clearly, equation O.14 has roots at $b = n \cdot \pi/2$, n any integer. The derivative -

$$\frac{d}{db}\left(\frac{f}{g}\right)(b) = \frac{f`(b)g(b) - f(b)g`(b)}{(g(b))^2}$$

$$\frac{(k-m)(cos^2(b) - sin^2(b))g(b) + cos(b)sin(b)(k-m)^2 sin(2b)}{(k \cdot cos^2(b) + m \cdot sin^2(b))^2}$$

is zero for $b = n \cdot \pi/4$, n any odd integer, confirming the incidence of maxima/minima of tanh(x).

Exponential (exp)

```
apl>Gz#exp x;a;b;c
[1]      z#(*a#9Ox)X(2Ob)+0j1X1Ob#11Ox
[2]      G
apl>(L#*x) = exp x#0j1+1 2 3

1.468694j2.287355 3.992324j6.217676 10.85226j16.90140

1 1 1
```

Assume that $x = COMPLEX(a, b)$ produces the complex number $a+bi$, the following mathematical expression is equivalent to the above CAPLIB2 statement for the *exp* user-defined function:

$$EXP(x) = e * *a * COMPLEX(COS(b), SIN(b)) \tag{O.15}$$

Figure P.24 on page 459 shows the exponential function for real numbers, and Figures P.25 and P.26 on pages 460 and 461 show graphs respectively for the real and imaginary coefficients of the output of the exponential function, given complex numbers as input, varying the real coefficient of the input, holding the imaginary coefficient to a constant. All of the graphs have the appearance of an exponential functions consistent with the definition in Equation O.15 above.

Figures P.27 and P.28 on pages 462 and 470 show graphs respectively for the real and imaginary coefficients of the output of the exponential function, given complex numbers as input, varying the imaginary coefficient of the input, holding the real coefficient to a constant. Again, consistent with Equation O.15, the graphs respectively behave like the **cosine** and **sine**, with the amplitude determined by the real coefficient of the input.

Arcsine for Real Numbers

Strictly speaking, the **sine** function for real numbers has no inverse since it is periodic. By restricting the domain of the inverse to $[-\pi/2, \pi/2]$ however, we get a workable function, continuous and differentiable. For any real number **y** from **O-.5** to **O.5** ($-\pi/2$ to $\pi/2$) inclusive, the CAPLIB2 expression **y=-1O1Oy** produces 1, and 0 otherwise:

```
apl>y#O(_‘Rt),0,t#.25 .5 .6 1 1.1 1.9 2 2.1 2.9 3 3.1
apl>L#v#y=z#-1Ox#1O

0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0
```

```
apl>(D/u),(S/u#v/y) " Range of y where y=-1O1Oy
```

```
-1.570796 1.570796
```

In those cases where the above produced 0, the difference from the answer to **-1O1Oy** and **y** is always some factor of π. So a way to get the inverse of **1Oy** for all **y** is to remember what multiple of π the original **y** value had:

```
apl>y=((-1*|n)Xz)+On#(Xy)XD((|y)+0.5)%O1
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Arccosine for Real Numbers

Like the **sine** function, **cosine** is periodic, so the domain of its inverse must be restricted to $0 \leq \arccos \leq \pi$ in order to get a continuous and differentiable function:

```
apl>y#O(_‘Rt),0,t#.25 .5 .6 1 1.1 1.9 2 2.1 2.9 3 3.1
apl>L#v#y=z#-2Ox#2Oy
```

```
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0
```

```
apl>(D/u),(S/u#v/y) " Range of y where y=-2O2Oy
```

```
0 3.141593
```

```
apl>y=z+(_((2Xz)_pi)X2|n)+On#Dy%pi#O1
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Arctangent for Real Numbers

Like the **sine** and **cosine** functions, **tangent** is periodic, so the domain of its inverse must be restricted to get a continuous and differentiable function. Since the tangent is undefined for $\pm\pi/2$ the domain of the inverse is the open interval $(-\pi/2, \pi/2)$:

```
apl>t#.25 .499 .6 1 1.1 1.9 2 2.1 2.9 3 3.1
apl>L#v#y=z#-3Ox#3Oy#O(_‘Rt),0,t
```

```
0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0
```

```
apl>(D/u),(S/u#v/y) " Range of y where y=-303Oy

-1.567655 1.567655

apl>y=z+OD(y+0.5)%pi#O1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Arcsinh (Inverse Hyperbolic Sine) for Real Numbers

Here's the mathematical expression used to implement the inverse hyperbolic sine function in CAPLIB2:

$$asinh(x) = log_e(x + \sqrt{x^2 + 1}) \tag{O.16}$$

In theory the domain of the inverse hyperbolic sine for real numbers is $(-\infty, \infty)$, but computer imprecision doesn't allow for accurate answers involving larger magnitudes:

```
apl>Lct#1e-10
apl>y#O(_‘Rt),0,t#.25 .5 .6 1 1.1 1.9 2 2.1 2.9 3 3.1 5
apl>L#v#y=z#-5Ox#5Oy

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

apl>(D/u),(S/u#v/y) " Range of y where y=-505Oy

-15.7 15.7

apl>v,x,y,z,[1.5]y_z

1 -3317812 -15.708 -15.708 0
1    -8483  -9.739  -9.739 0
1    -6196  -9.425  -9.425 0
1    -4525  -9.111  -9.111 0
1     -367  -6.597  -6.597 0
1     -268  -6.283  -6.283 0
1     -196  -5.969  -5.969 0
1      -16  -3.456  -3.456 0
1      -12  -3.142  -3.142 0
1       -3  -1.885  -1.885 0
1       -2  -1.571  -1.571 0
```

```
1       -1  -0.785  -0.785 0
1        0   0.000   0.000 0
1        1   0.785   0.785 0
1        2   1.571   1.571 0
1        3   1.885   1.885 0
1       12   3.142   3.142 0
1       16   3.456   3.456 0
1      196   5.969   5.969 0
1      268   6.283   6.283 0
1      367   6.597   6.597 0
1     4525   9.111   9.111 0
1     6196   9.425   9.425 0
1     8483   9.739   9.739 0
1  3317812  15.708  15.708 0
```

Arccosh (Inverse Hyperbolic Cosine) for Real Numbers

Here's the mathematical expression used to implement the inverse hyperbolic cosine function in CAPLIB2:

$$acosh(x) = log_e(x + \sqrt{x^2 - 1}) \qquad \text{(O.17)}$$

Since **(6O_x)=6Ox** produces 1 for any x, the inverse is multi-valued, so **-6Ox** returns only the positive answer:

```
apl>y#O(_‘Rt),0,t#.25 .5 .6 1 1.1 1.9 2 2.1 2.9 3 3.1
apl>L#v#y=z#-6Ox#6Oy

0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1

apl>v,x,y,z,[1.5]y_z

0 8482.7507  -9.738937  9.738937 -19.477874
0 6195.8239  -9.424778  9.424778 -18.849556
0 4525.4465  -9.110619  9.110619 -18.221237
0  366.5736  -6.597345  6.597345 -13.194689
0  267.7468  -6.283185  6.283185 -12.566371
0  195.5636  -5.969026  5.969026 -11.938052
0   15.8568  -3.455752  3.455752  -6.911504
0   11.5920  -3.141593  3.141593  -6.283185
0    3.3689  -1.884956  1.884956  -3.769911
0    2.5092  -1.570796  1.570796  -3.141593
0    1.3246  -0.785398  0.785398  -1.570796
```

```
1    1.0000   0.000000  0.000000   0.000000
1    1.3246   0.785398  0.785398   0.000000
1    2.5092   1.570796  1.570796   0.000000
1    3.3689   1.884956  1.884956   0.000000
1   11.5920   3.141593  3.141593   0.000000
1   15.8568   3.455752  3.455752   0.000000
1  195.5636   5.969026  5.969026   0.000000
1  267.7468   6.283185  6.283185   0.000000
1  366.5736   6.597345  6.597345   0.000000
1 4525.4465   9.110619  9.110619   0.000000
1 6195.8239   9.424778  9.424778   0.000000
1 8482.7507   9.738937  9.738937   0.000000
```

Arctanh (Inverse Hyperbolic Tangent) for Real Numbers

Here's the mathematical expression used to implement the inverse hyperbolic tangent function in CAPLIB2:

$$atanh(x) = \frac{1}{2}log_e(\frac{1+x}{1-x}) \tag{O.18}$$

Since the range of the hyperbolic tangent for real numbers is $(-1, 1)$ for a domain of $(-\infty, \infty)$, in theory the domain of the inverse hyperbolic tangent for real numbers should also be $(-1, 1)$. In practice, digital computers are too imprecise for anything close to -1 or 1:

```
apl>&/v#y=z#-7Ox#7Oy#(_‘Rt),0,t#.5 1.5 2 3 4 10 15

0

apl>(D/u),(S/u#v/y) " Range of y where y=-7O7Oy

-4 4

apl>v,x,y,z,[1.5]y_z

0 -1.0000000 -15.0 -14.99992 -0.00008319
0 -1.0000000 -10.0 -10.00000  0.00000000
1 -0.9993293  -4.0  -4.00000  0.00000000
1 -0.9950548  -3.0  -3.00000  0.00000000
1 -0.9640276  -2.0  -2.00000  0.00000000
1 -0.9051483  -1.5  -1.50000  0.00000000
1 -0.4621172  -0.5  -0.50000  0.00000000
```

```
1  0.0000000   0.0   0.00000  0.00000000
1  0.4621172   0.5   0.50000  0.00000000
1  0.9051483   1.5   1.50000  0.00000000
1  0.9640276   2.0   2.00000  0.00000000
1  0.9950548   3.0   3.00000  0.00000000
1  0.9993293   4.0   4.00000  0.00000000
0  1.0000000  10.0  10.00000  0.00000000
0  1.0000000  15.0  14.99992  0.00008319
```

Arcsine for Complex Numbers

This expression defines the arcsine for complex numbers:

$$-i \cdot asinh(i \cdot x) \tag{O.19}$$

Here's an equivalent user-defined **asin** function:

```
apl>Lfx 'z#asin x' 'z#_0j1X-5O0j1Xx'

asin

apl>&/(-1Ox) = asin x#1Oy

1
```

The CAPLIB2 **asin** is a true inverse, i.e. **(y=-1O1Oy)** produces 1, only if **9Oy** ranges between $\pm\pi/2$, and **11Oy** between $\pm 2\pi$:

```
apl>y#,(wX0j1)J.+w#(_‘R1Uw),w#O2X(-1+Ipts+1)%pts#8
apl>v#y=z#-1Ox#1Oy
apl>(D/u),(S/u#v/9Oy) " Range of 9Oy where y=-1O1Oy

-1.57 1.57

apl>(D/u),(S/u#v/11Oy) " Range of 11Oy where y=-1O1Oy

-6.28 6.28
```

Arccosine for Complex Numbers

This expression defines the arccosine for complex numbers:

$$-i \cdot log_e \left(x - i \cdot \sqrt{1 - x^2} \right) \tag{O.20}$$

Here's an equivalent user-defined **acos** function:

```
apl>Lfx 'z#acos x' 'z#_0j1X@x_0j1X(1_x*2)*.5'

acos

apl>&/(-20x) = acos x#20y

1
```

The CAPLIB2 **acos** is a true inverse, i.e. **(y=-2O2Oy)** produces 1, only if **9Oy** ranges between $\pm\pi$, and **11Oy** between $\pm 2\pi$:

```
apl>y#,(wX0j1)J.+w#(_`R1Uw),w#02X(-1+Ipts+1)%pts#8
apl>v#y=z#-20x#20y
apl>(D/u),(S/u#v/90y) " Range of 90y where y=-2020y

-3.14 3.14

apl>(D/u),(S/u#v/110y) " Range of 110y where y=-2020y

-6.28 6.28
```

Arctangent for Complex Numbers

This expression defines the arctangent for complex numbers:

$$\frac{1}{2} \cdot i \cdot log_e \left(\frac{1 - i \cdot x}{1 + i \cdot x} \right) \tag{O.21}$$

Here's an equivalent user-defined **atan** function:

```
apl>Lfx 'z#atan x' 'z#(0j1X@(1_0j1Xx)%1+0j1Xx)%2'

atan

apl>&/(-30x) = atan x#30y

1
```

The CAPLIB2 **atan** is a true inverse, i.e. **(y=-3O3Oy)** produces 1, only if **9Oy** ranges between $\pm\pi/2$, and **11Oy** between $\pm 2\pi$:

```
apl>y#,(wX0j1)J.+w#(_`R1Uw),w#02X(-1+Ipts+1)%pts#8
apl>v#y=z#-30x#30y
apl>(D/u),(S/u#v/90y) " Range of 90y where y=-3030y
```

```
-1.57 1.57

apl>(D/u),(S/u#v/11Oy) " Range of 11Oy where y=-3O3Oy

-6.28 6.28
```

Arcsinh (Inverse Hyperbolic Sine) for Complex Numbers

The expression for the inverse hyperbolic sine function in CAPLIB2 for complex numbers is identical to that used for real numbers, equation O.16. Here's an equivalent CAPLIB2 expression, where **y** can be any real or complex number:

```
apl>Lfx 'z#asinh x' 'z#@x+(1+x*2)*.5'

asinh

apl>&/(-5Ox) = asinh x#5Oy

1
```

I tested a variety of values, and found it to be a true inverse, i.e. **(y=-5O5Oy)** produces 1, only if **9Oy** ranges between $\pm 2\pi$, and **11Oy** between $\pm \pi/2$:

```
apl>y#,(wX0j1)J.+w#(_‘R1Uw),w#O2X(-1+Ipts+1)%pts#8
apl>&/v#y=z#-5Ox#5Oy

0

apl>(D/u),(S/u#v/9Oy) " Range of 9Oy where y=-5O5Oy

-6.283 6.283

apl>(D/u),(S/u#v/11Oy) " Range of 11Oy where y=-5O5Oy

-1.57 1.57
```

Arccosh (Inverse Hyperbolic Cosine) for Complex Numbers

The expression for the inverse hyperbolic cosine function in CAPLIB2 for complex numbers is identical to that used for real numbers, equation O.17.

Here's an equivalent CAPLIB^2 expression, where **y** can be any real or complex number:

```
apl>Lfx 'z#acosh x' 'z#@x+(-1+x*2)*.5'

acosh

apl>&/(-6Ox) = acosh x#6Oy

1
```

I tested a variety of values, and found it to be a true inverse, i.e. **(y=-6O6Oy)** produces 1, only if **9Oy** ranges between $\pm 2\pi$, and **11Oy** between $\pm\pi$:

```
apl>y#,(wXOj1)J.+w#(_‘R1Uw),w#O2X(-1+Ipts+1)%pts#8
apl>&/v#y=z#-6Ox#6Oy

0

apl>(D/u),(S/u#v/9Oy) " Range of 9Oy where y=-6O6Oy

-6.283 6.283

apl>(D/u),(S/u#v/11Oy) " Range of 11Oy where y=-6O6Oy

-3.142 3.142
```

Arctanh (Inverse Hyperbolic Tangent) for Complex Numbers

The expression for the inverse hyperbolic tangent function in CAPLIB^2 for complex numbers is identical to that used for real numbers, equation O.18. Here's an equivalent CAPLIB^2 expression, where **y** can be any real or complex number:

```
apl>Lfx 'z#atanh x' 'z#.5X@(1+x)%1_x'

atanh

apl>&/(-7Ox) = atanh x#7Oy

1
```

I tested a variety of values, and found it to be a true inverse, i.e. **(y=-7O7Oy)** produces 1, only if **9Oy** ranges between $\pm 2\pi$, and **11Oy** between $\pm\pi/2$:

```
apl>y#,(wX0j1)J.+w#(_‘R1Uw),w#O2X(-1+Ipts+1)%pts#8
apl>&/v#y=z#-7Ox#7Oy

0

apl>(D/u),(S/u#v/9Oy) " Range of 9Oy where y=-7O7Oy

-6.283 6.283

apl>(D/u),(S/u#v/11Oy) " Range of 11Oy where y=-7O7Oy

-1.57 1.57
```

Equivalent Trigonometric Expressions

Here's a list of CAPLIB^2 expressions that all produce 1*s*. They're all translations of fundamental identities from the "Handbook of Mathematical Tables and Formulas" (see Reference [4]). I ran all of these through the CAPLIB^2 interpreter, as a test to ensure the implementation is correct.

This list starts with some simple defined functions to make the expressions that follow more readable, and with the assignment of a few variables. These statements were submitted to the CAPLIB^2 interpreter to ensure the results are as expected:

```
" <----------------------------------------------->
Lfx 'z # sin x'  'z # 1 O x'
Lfx 'z # cos x'  'z # 2 O x'
Lfx 'z # tan x'  'z # 3 O x'
Lfx 'z # sinh x' 'z # 5 O x'
Lfx 'z # cosh x' 'z # 6 O x'
Lfx 'z # tanh x' 'z # 7 O x'
Lfx 'z # asin x'  'z # -1 O x'
Lfx 'z # acos x'  'z # -2 O x'
Lfx 'z # atan x'  'z # -3 O x'
Lfx 'z # asinh x' 'z # -5 O x'
Lfx 'z # acosh x' 'z # -6 O x'
Lfx 'z # atanh x' 'z # -7 O x'
Lct#1e-8
i#0j1
```

```
n#4
r # .5 3j4
s # .2 4j5
" <-all the following expressions produce only 1s-->
(sin r) = ((*iXr)_1%*iXr)%2Xi
(cos r) = ((*iXr)+1%*iXr)%2
(tan r) = (sin r) % cos r
(asin r) = _iX@(iXr)+(1_r*2)*.5
(acos r) = _iX@r+iX(1_r*2)*.5
(atan r) = (iX@(1_iXr)%1+iXr)%2
(sinh r) = ((*r)_1%*r)%2
(cosh r) = ((*r)+1%*r)%2
(tanh r) = (t_u)%t+u#1%t#*r
(asinh r) = @r+(1+r*2)*.5
(acosh r) = @(r+-4Or)
(acosh r) = @r+(-1+r*2)*.5
(atanh r) = (@(1+r)%(1_r))%2
(sin 2Xr) = 2 X (sin r) X (cos r)
(cos 2Xr) = -1 + 2 X (cos r)*2
(cos 2Xr) = 1 _ 2 X (sin r)*2
(cos 2Xr) = ((cos r)*2) _ ((sin r)*2)
(tan 2Xr) = (2 X tan r) % 1 _ (tan r)*2
(sin 3Xr) = (3 X sin r) _ 4 X (sin r)*3
(cos 3Xr) = (4 X (cos r)*3) _ 3 X cos r
(sin nXr) = ((2Xsin rXn_1)X(cos r))_sin rXn_2
(cos nXr) = ((2Xcos rXn_1)X(cos r))_cos rXn_2
(sin r+s) = ((sin r)X(cos s))+(cos r)Xsin s
(sin r_s) = ((sin r)X(cos s))_(cos r)Xsin s
(cos r+s) = ((cos r)X(cos s))_(sin r)Xsin s
(cos r_s) = ((cos r)X(cos s))+(sin r)Xsin s
(tan r+s) = ((tan r)+tan s)%1_(tan r)Xtan s
(tan r_s) = ((tan r)_tan s)%1+(tan r)Xtan s
((sin r)+sin s) = 2X(sin(r+s)%2)Xcos(r_s)%2
((sin r)_sin s) = 2X(cos(r+s)%2)Xsin(r_s)%2
((cos r)+cos s) = 2X(cos(r+s)%2)Xcos(r_s)%2
((cos r)_cos s) =_2X(sin(r+s)%2)Xsin(r_s)%2
((sin r)*2) = (1_cos 2Xr)%2
((cos r)*2) = (1+cos 2Xr)%2
((sin r)*3) = ((3Xsin r)_sin 3Xr)%4
((cos r)*3) = ((3Xcos r)+cos 3Xr)%4
((sin r)Xsin s) = ((cos r_s)%2)_(cos r+s)%2
((cos r)Xcos s) = ((cos r_s)%2)+(cos r+s)%2
```

```
((sin r)Xcos s) = ((sin r+s)%2)+(sin r_s)%2
(asinh r) = @r+(1+r*2)*.5
(acosh r) = @r+(-1+r*2)*.5
(atanh r) = (@(1+r)%1_r)%2
(sinh _r) = _sinh r
(cosh _r) = cosh r
(tanh _r) = _tanh r
1 = ((cosh r)*2)_(sinh r)*2
(sinh r+s) = ((sinh r)Xcosh s)+(cosh r)Xsinh s
(sinh r_s) = ((sinh r)Xcosh s)_(cosh r)Xsinh s
(cosh r+s) = ((cosh r)Xcosh s)+(sinh r)Xsinh s
(cosh r_s) = ((cosh r)Xcosh s)_(sinh r)Xsinh s
(tanh r+s) = ((tanh r)+tanh s)%1+(tanh r)Xtanh s
(tanh r_s) = ((tanh r)_tanh s)%1_(tanh r)Xtanh s
(sinh 2Xr) = 2X(sinh r)Xcosh r
(cosh 2Xr) = ((cosh r)*2)+(sinh r)*2
(2X(sinh r%2)*2) = -1+cosh r
(2X(cosh r%2)*2) = 1+cosh r
(sin r) = _iXsinh iXr
(cos r) = cosh iXr
(tan r) = _iXtanh iXr
(sin iXr) = iXsinh r
(cos iXr) = cosh r
(tan iXr) = iXtanh r
(sinh iXr) = iXsin r
(cosh iXr) = cos r
(tanh iXr) = iXtan r
(sinh r+iXs) = ((sinh r)Xcos s)+iX(cosh r)Xsin s
(sinh r_iXs) = ((sinh r)Xcos s)_iX(cosh r)Xsin s
(cosh r+iXs) = ((cosh r)Xcos s)+iX(sinh r)Xsin s
(cosh r_iXs) = ((cosh r)Xcos s)_iX(sinh r)Xsin s
(sinh r+2XiXO1) = sinh r
(cosh r+2XiXO1) = cosh r
(sinh r+iXO1) = _sinh r
(cosh r+iXO1) = _cosh r
(sinh r+.5XiXO1) = iXcosh r
(cosh r+.5XiXO1) = iXsinh r
(*r) = (cosh r)+sinh r
(1%*r) = (cosh r)_sinh r
(*iXr) = (cos r)+iXsin r
(1%*iXr) = (cos r)_iXsin r
r = (|r)X*iXacos (9Or)%|r
```

```
r = (|r)X*iXasin (11Or)%|r
(((cos r)+iXsin r)*n) = (cos nXr)+iXsin nXr
1 = ((cos t)+iXsin t#(O2)X-1+In)*n
((sinh r)+sinh s) = 2X(sinh.5Xr+s)Xcosh.5Xr_s
((sinh r)_sinh s) = 2X(cosh.5Xr+s)Xsinh.5Xr_s
((cosh r)+cosh s) = 2X(cosh.5Xr+s)Xcosh.5Xr_s
((cosh r)_cosh s) = 2X(sinh.5Xr+s)Xsinh.5Xr_s
" <--------------------------------------------------->
```

Here are CAPLIB2 statements which test alternative methods for obtaining the answers to the circle functions. Again, these statements were submitted to the CAPLIB2 interpreter to ensure the results are as expected, and they rely for readability on defined functions for **sin, cos**, etc., shown previously:

```
Lfx 'getxy' 'x#9 O r' 'y#11 O r'
i#0j1 " square root of -1
src#'z#sina r;x;y' 'getxy'
Lfx src,‘Z'z#((sin x)Xcosh y)+iX(cos x)Xsinh y'
Lfx 'z#sinb r;s' 'z#(s_1%s#*iXr)%2Xi'
Lfx 'z#sinc r;s' 'z#_i X sinh iXr'
src#'z#cosa r;x;y' 'getxy'
Lfx src,‘Z'z#((cos x)Xcosh y)_iX(sin x)Xsinh y'
Lfx 'z#cosb r;s' 'z#(s+1%s#*iXr)%2'
Lfx 'z#cosc r;s' 'z#cosh iXr'
src#'z#tana r;x;y' 'getxy'
srd#'z#((sin x)+iX(cos x)X(tanh y))'
srd#srd,'%(cos x)_iX(sin x)Xtanh y'
Lfx src,‘Zsrc srd
Lfx 'z#tanb r' 'z#(sin r)%(cos r)'
Lfx 'z#tanc r' 'z#_iXtanh iXr'
src#'z#sinha r;x;y;s' 'getxy' 'y#yXs#Xy'
Lfx src,‘Z'z#((sinh x)Xcos y)+sXiX(cosh x)Xsin y'
Lfx 'z#sinhb r' 'z#_iXsin iXr'
Lfx 'z#sinhc r;x' 'x#r%2' 'z#2X(sinh x)Xcosh x'
src#'z#cosha r;x;y;s' 'getxy' 'y#yXs#Xy'
Lfx src,‘Z'z#((cosh x)Xcos y)+sXiX(sinh x)Xsin y'
Lfx 'z#coshb r' 'z#cos iXr'
Lfx 'z#coshc r;x' 'x#r%2' 'z#((cosh x)*2)+(sinh x)*2'
src#'z#tanha r;x;y;s' 'getxy' 'y#iXyXs#Xy'
Lfx src,‘Z'z#((tanh x)+sXtanh y)%1+sX(tanh x)Xtanh y'
Lfx 'z#tanhb r;s;t' 'z#(t_s)%t+s#1%t#*r'
```

```
Lfx 'z#tanhc r' 'z#(sinh r)%cosh r'
Lfx 'z#tanhd r' 'z#_iXtan iXr'
Lfx 'z#asina r' 'z#_iXasinh iXr'
Lfx 'z#asinb r' 'z#_iX@(iXr)+(1_r*2)*.5'
Lfx 'z#acosa r' 'z#(.5XO1)_asin r'
Lfx 'z#acosb r' 'z#_iX@r+iX(1_r*2)*.5'
Lfx 'z#atana r' 'z#_iXatanh iXr'
Lfx 'z#atanb r' 'z#_iX@(1+iXr)%(1+r*2)*.5'
Lfx 'z#atanc r' 'z#(iX@(1_iXr)%(1+iXr))%2'
Lfx 'z#asinha r' 'z#_iXasin iXr'
Lfx 'z#asinhb r' 'z#@r+(1+r*2)*.5'
Lfx 'z#acosha r' 'z#@(r+-4Or)'
Lfx 'z#acoshb r' 'z#@r+(-1+r*2)*.5'
Lfx 'z#atanha r' 'z#_iXatan iXr'
Lfx 'z#atanhb r' 'z#(@(1+r)%(1_r))%2'
Lex 'demo'
Gname demo r;x;y;z
y#C`1x#((1Yname),`1' ',`Z`12`Xname),`1`Z' r'
$goodXI&/`Xy=1Yy
'oops, methods don''t agree on "',name,' ',(,Fr),'"'
y#F`1y
z#((-1Y`1R`1x)S-1Y`1R`1y)R`1'-'
' ',[1]x,[1]z,[.5]y
$0
good:
'all methods agree on "',(1`Xname),' ',(,Fr),'" = ',F1Yy
G
```

And here are the results:

```
apl>Lpp#5
apl>'sin'  'abc'  demo 3j4
all methods agree on "sin 3j4" =  3.8537j-27.017
apl>'cos'  'abc'  demo 3j4
all methods agree on "cos 3j4" =  -27.035j-3.8512
apl>'tan'  'abc'  demo 3j4
all methods agree on "tan 3j4" =  -0.00019j.9994
apl>'sinh' 'abc'  demo 3j4
all methods agree on "sinh 3j4" =  -6.5481j-7.6192
apl>'cosh' 'abc'  demo 3j4
all methods agree on "cosh 3j4" =  -6.5807j-7.5816
apl>'tanh' 'abcd' demo 3j4
all methods agree on "tanh 3j4" =  1.0007j.00491
```

```
apl>'asin' 'ab'   demo 3j4
all methods agree on "asin 3j4" =  .6340j2.3055
apl>'acos' 'ab'   demo 3j4
all methods agree on "acos 3j4" =  .9368j-2.3055
apl>'atan' 'abc'  demo 3j4
all methods agree on "atan 3j4" =  -0.12249j.15900
apl>'acosh' 'ab'  demo 3j4
all methods agree on "acosh 3j4" =  1.9684j.14848
apl>'atanh' 'ab'  demo 3j4
all methods agree on "atanh 3j4" =  .11750j-0.16088
apl>'asinh' 'ab'  demo 3j4
oops, methods don't agree on  " asinh ab  3j4"

 asinh  r       asinha r       asinhb r
 -------------- -------------- --------------
 1.9436j.13487  -1.9630j1.4412 1.9436j.13487
```

The **circular** functions *sin*, *cos*, and *tan* work on radians, which can be derived from degrees, and degrees from radians with expressions like these:

```
apl>F`1'sine of '    t ' degrees = ' (1O0(t#30)%180)

 sine of  30  degrees =   .5

apl>F`1'cosine of '  t ' degrees = ' (2O0(t#45)%180)

 cosine of  45  degrees =   .7071068

apl>F`1'arcsine of ' t ' = ' ((-1Ot#.5)X180%O1) ' degrees'

 arcsine of   .5  =  30  degrees

apl>F`1'arctan of '  t ' = ' ((-3Ot#1)X180%O1)  ' degrees'

 arctan of  1  =  45  degrees
```

Appendix P

Graphs of Trigonometric Functions

Figure P.1: Graph of y#1Ox

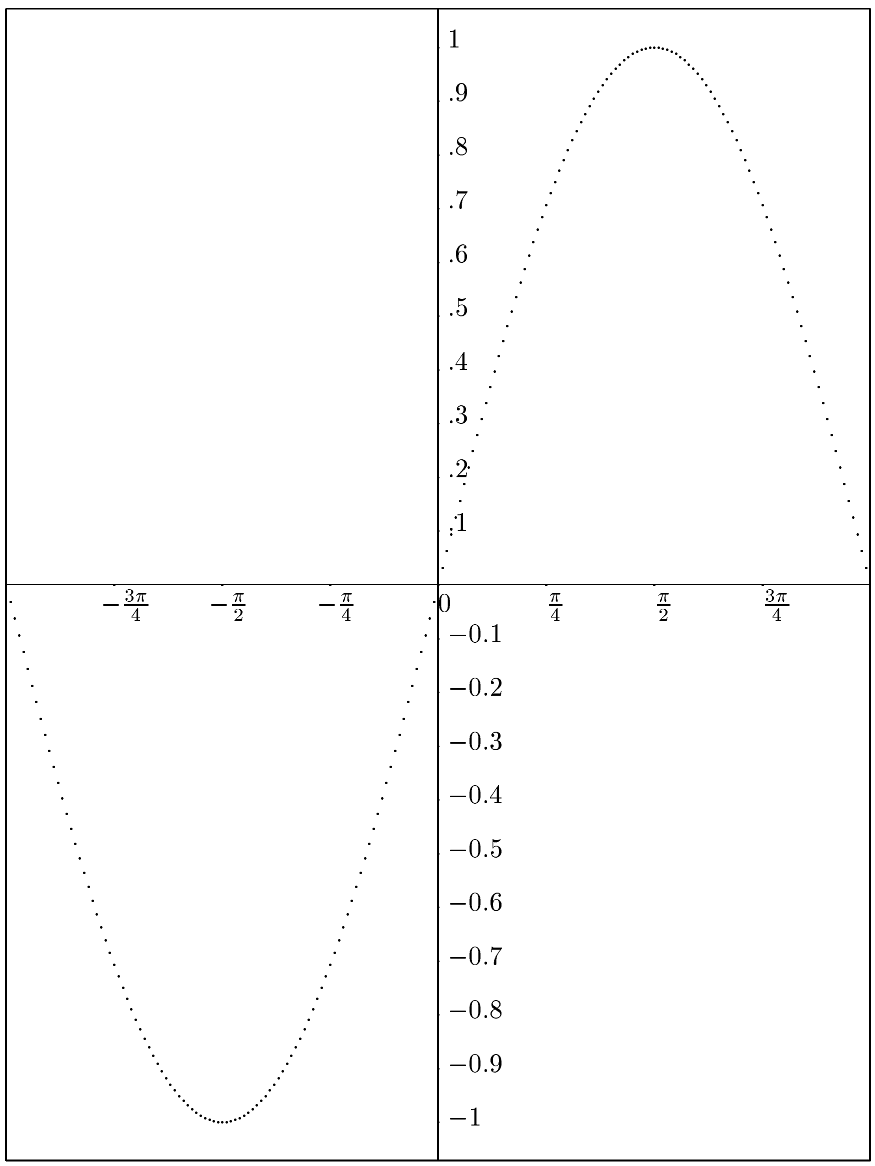


Figure P.2: Graph of y#2Ox

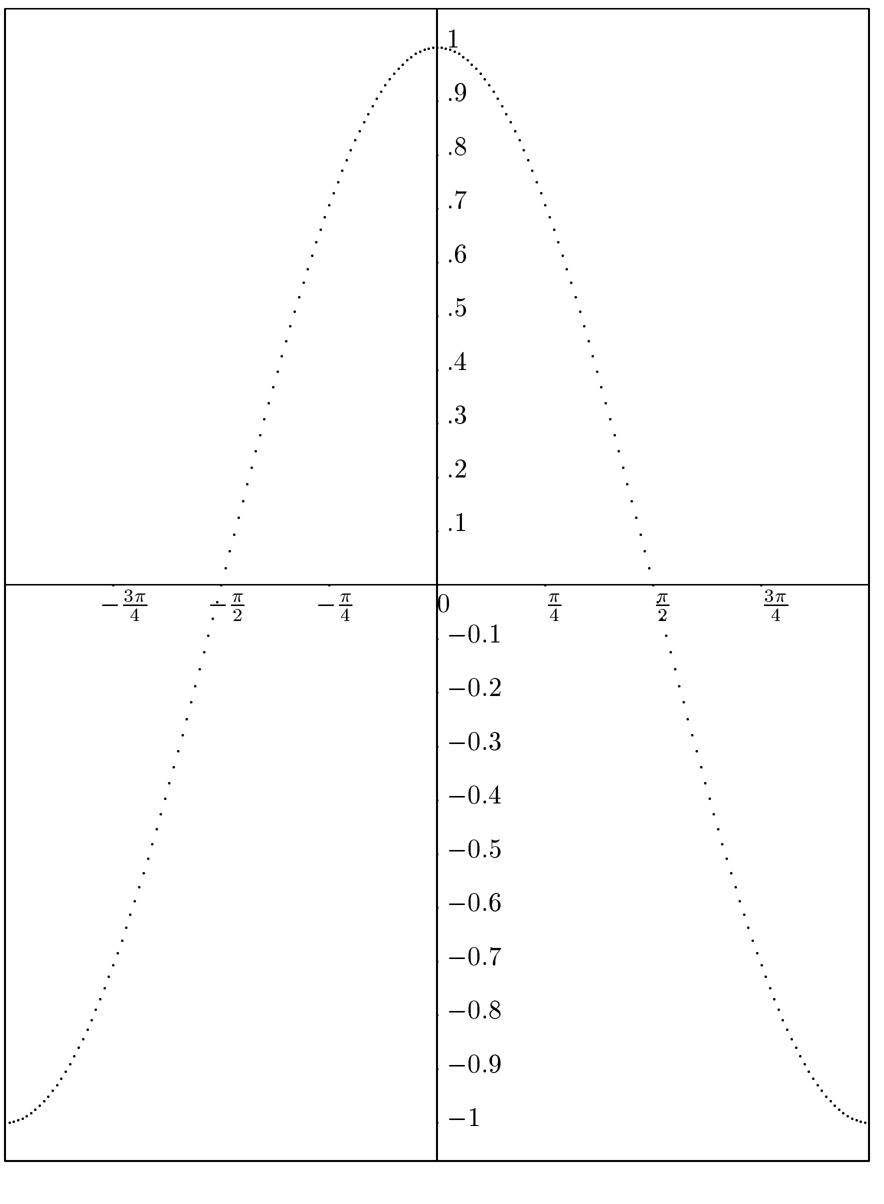


Figure P.3: Graph of y#3Ox

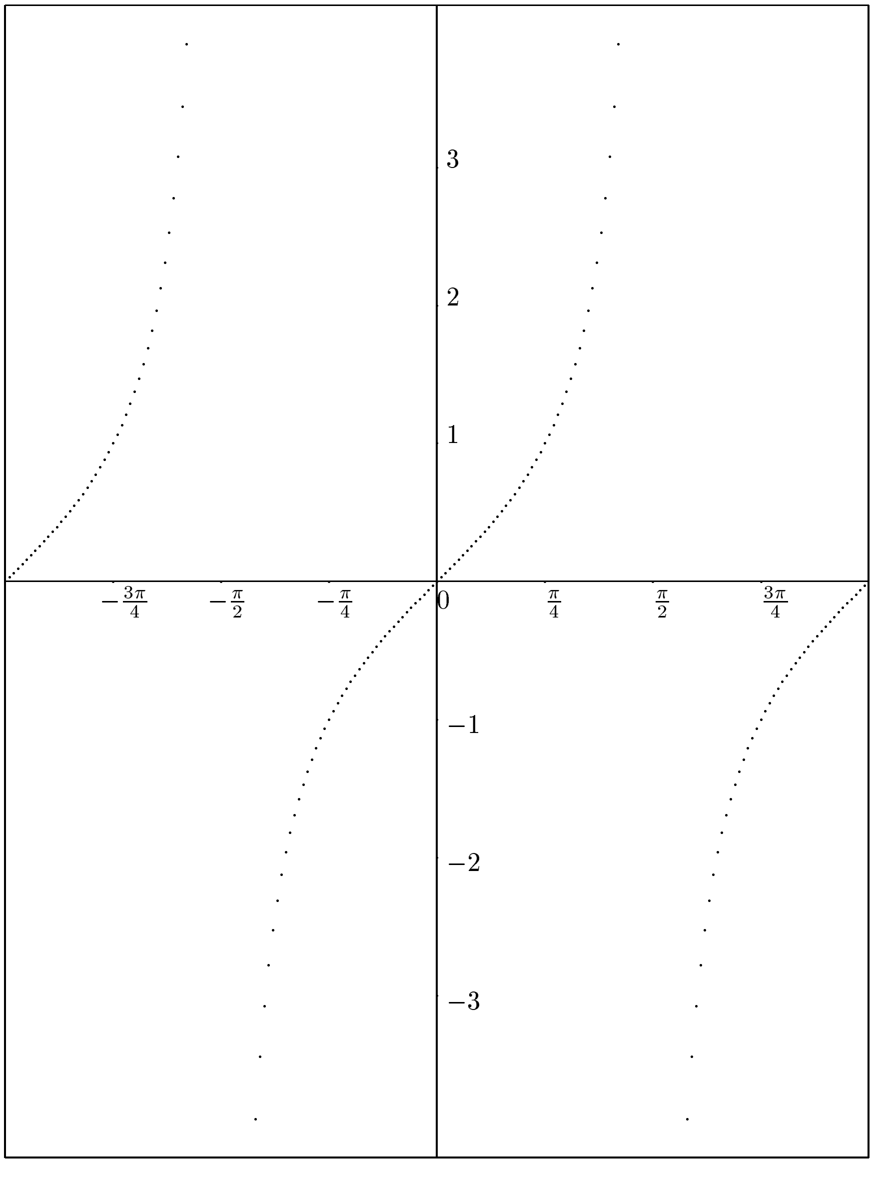


Figure P.4: Graphs of 5Ox and 6Ox

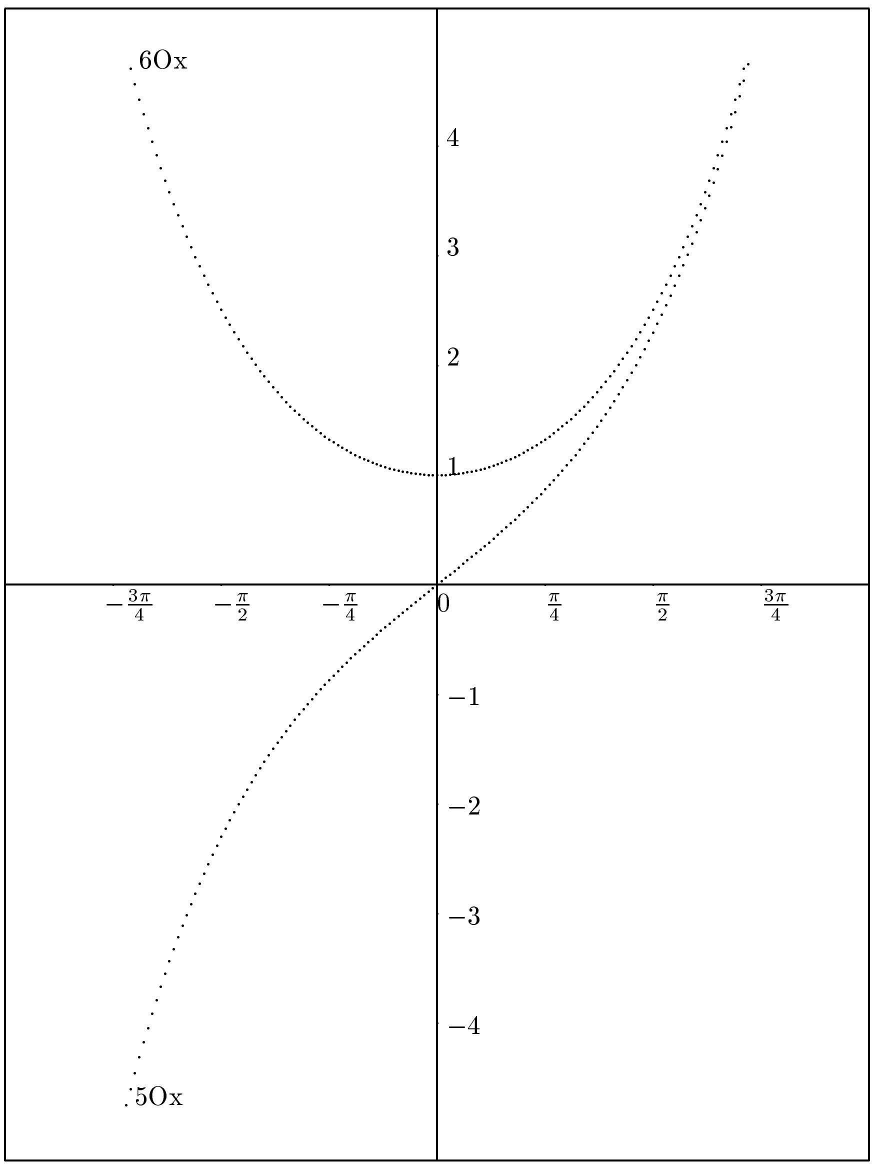


Figure P.5: Graph of y#9O1Ox+nX0j1

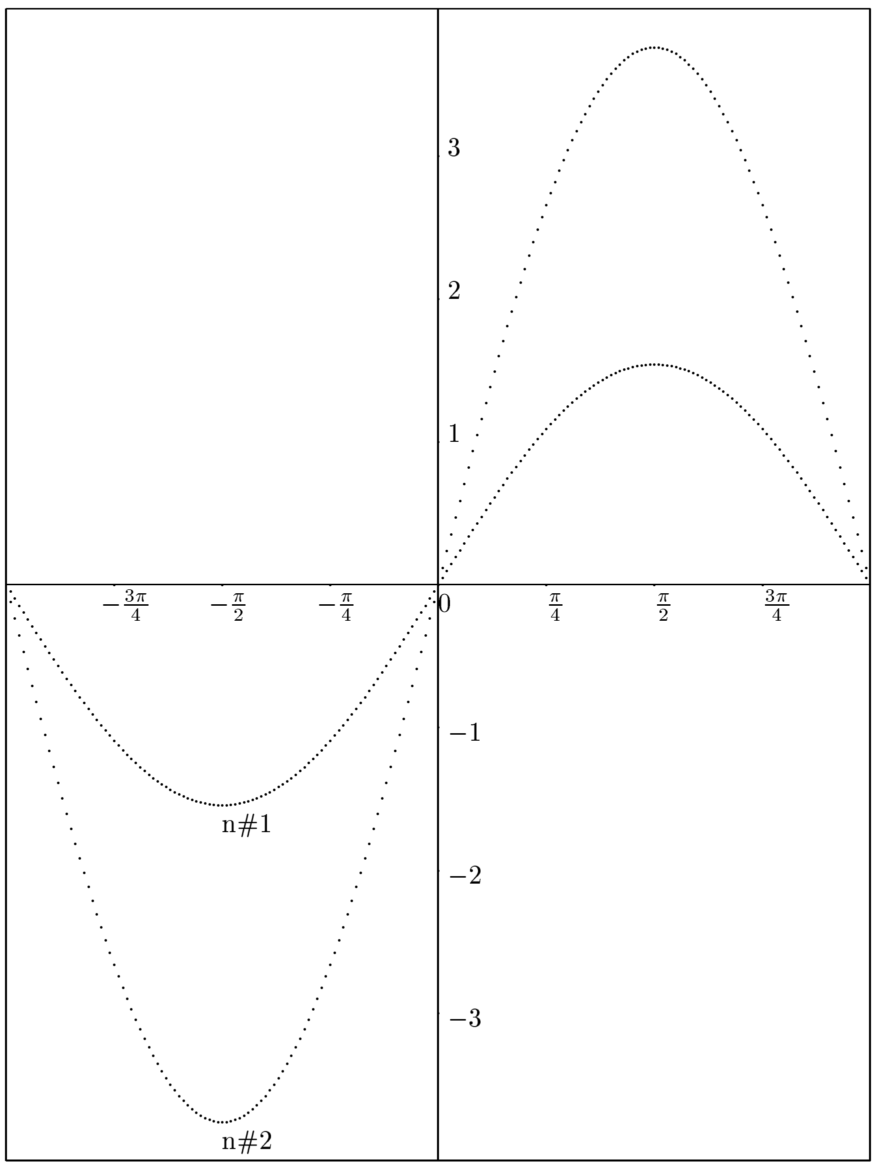


Figure P.6: Graph of y#11O1Ox+nX0j1

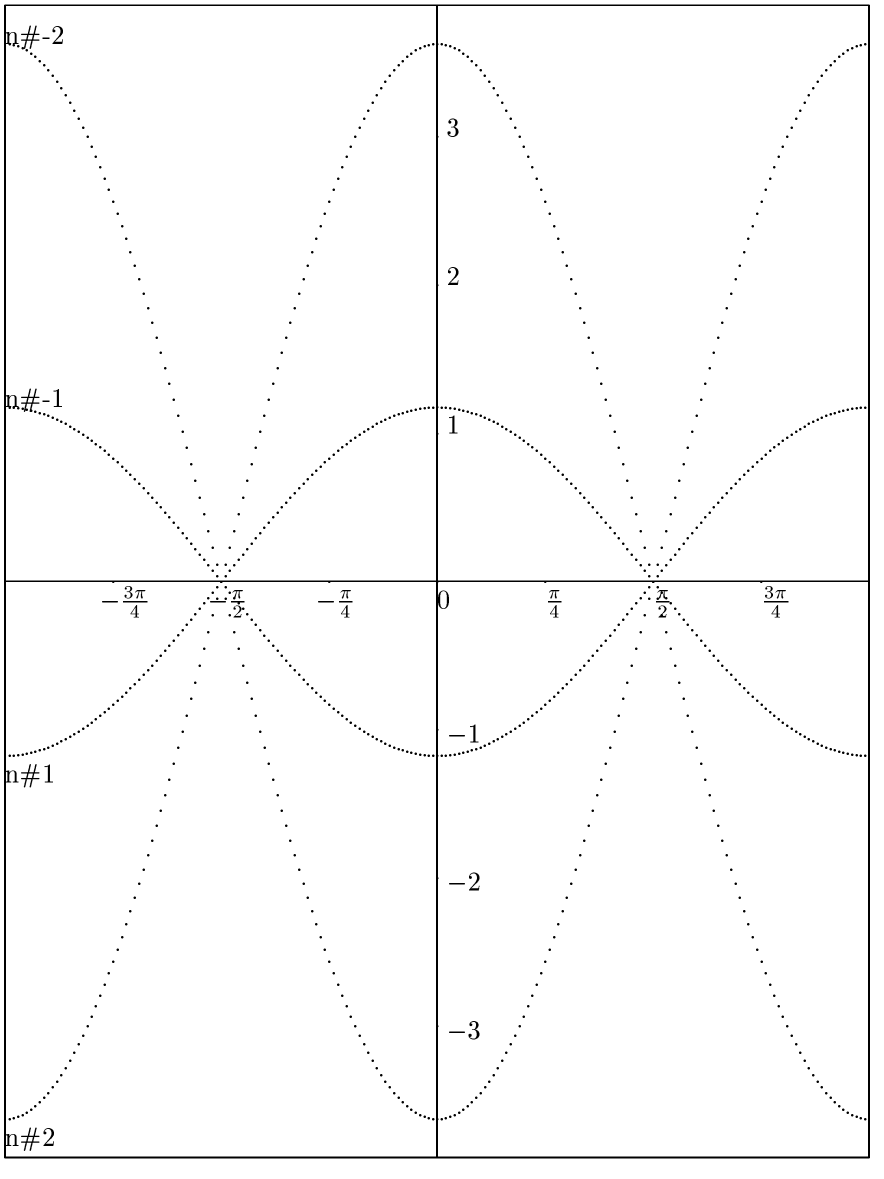


Figure P.7: Graph of y#9O2Ox+nX0j1

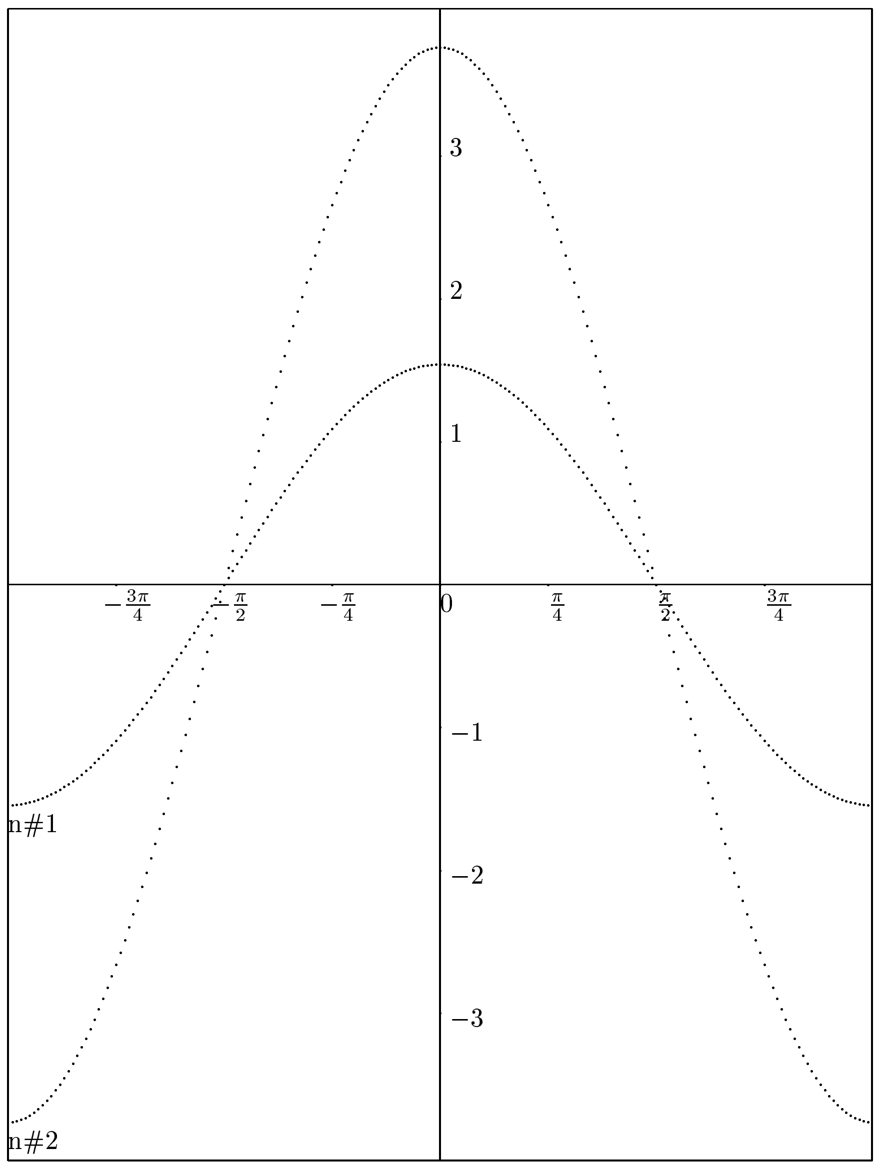


Figure P.8: Graph of y#11O2Ox+nX0j1

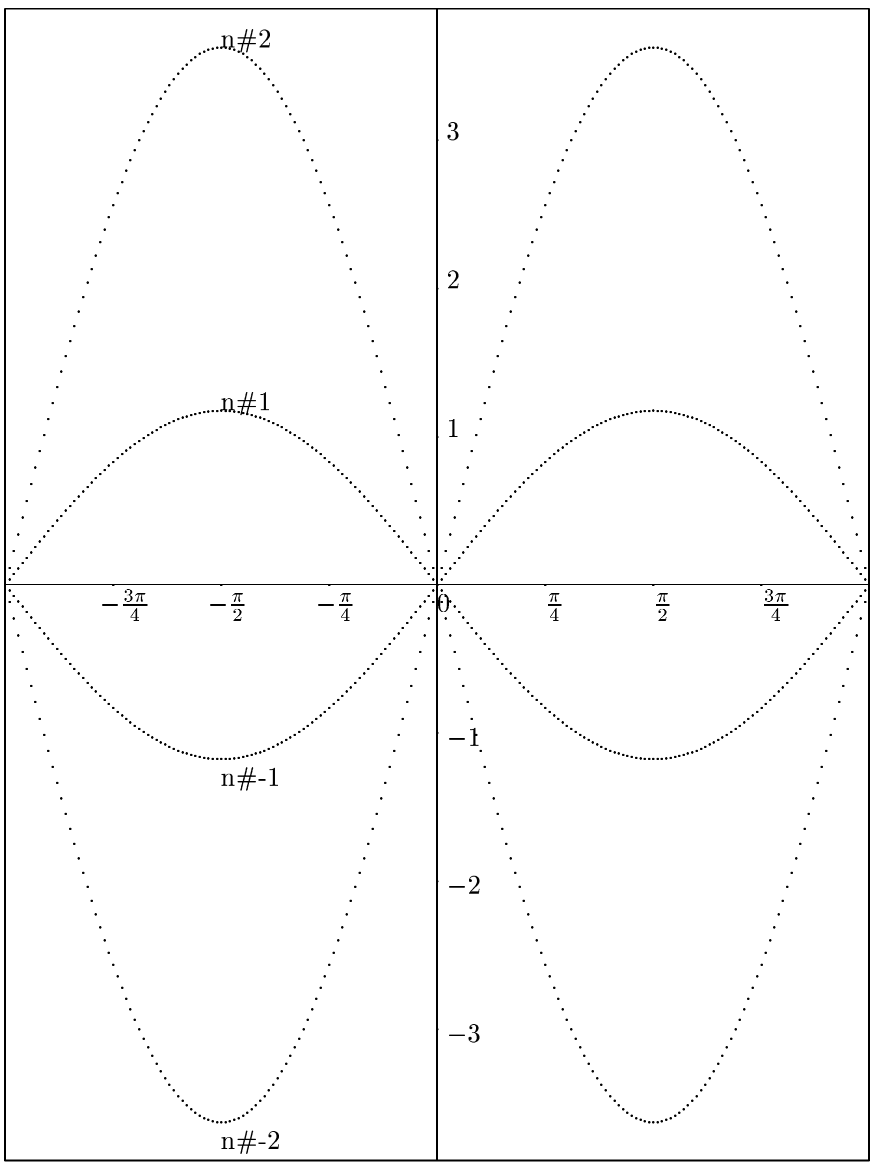


Figure P.9: Graph of y#9O3Ox+nX0j1

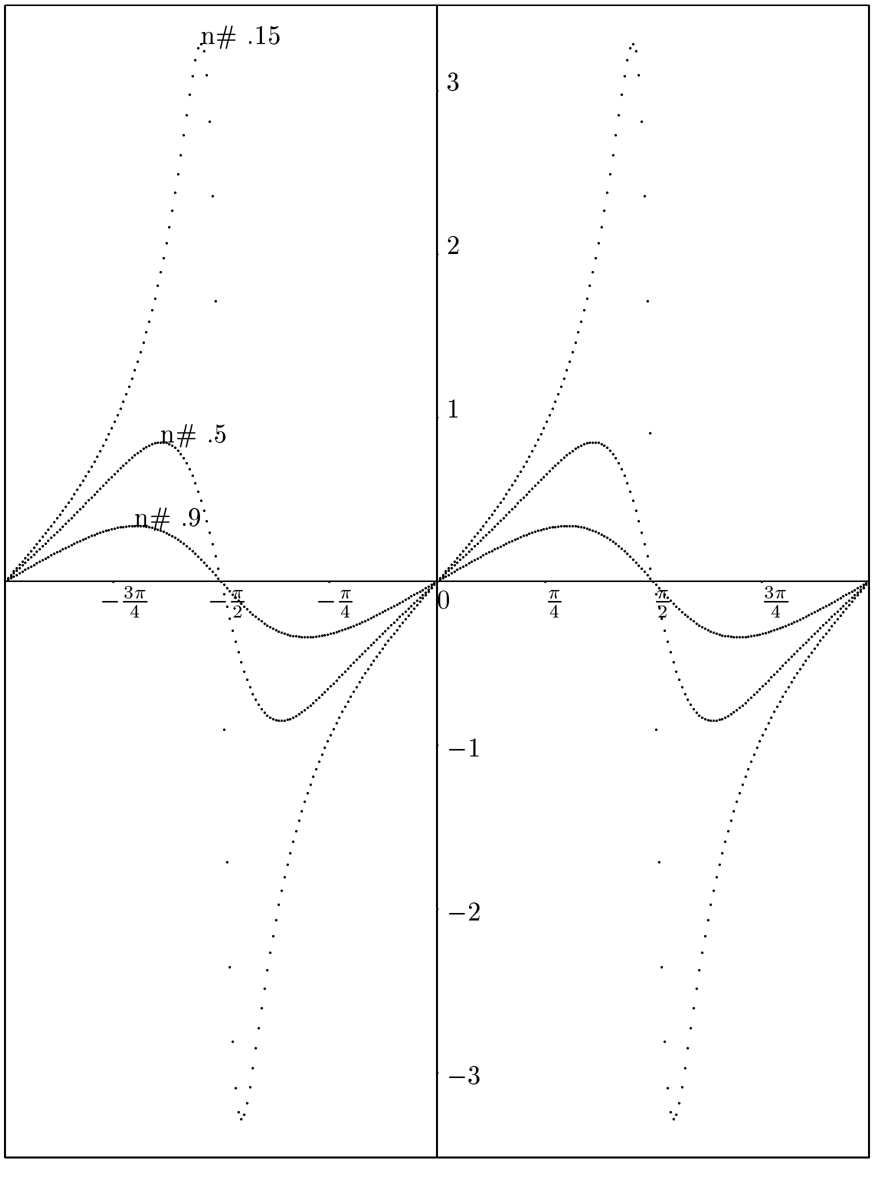


Figure P.10: Graph of y#11O3Ox+nX0j1

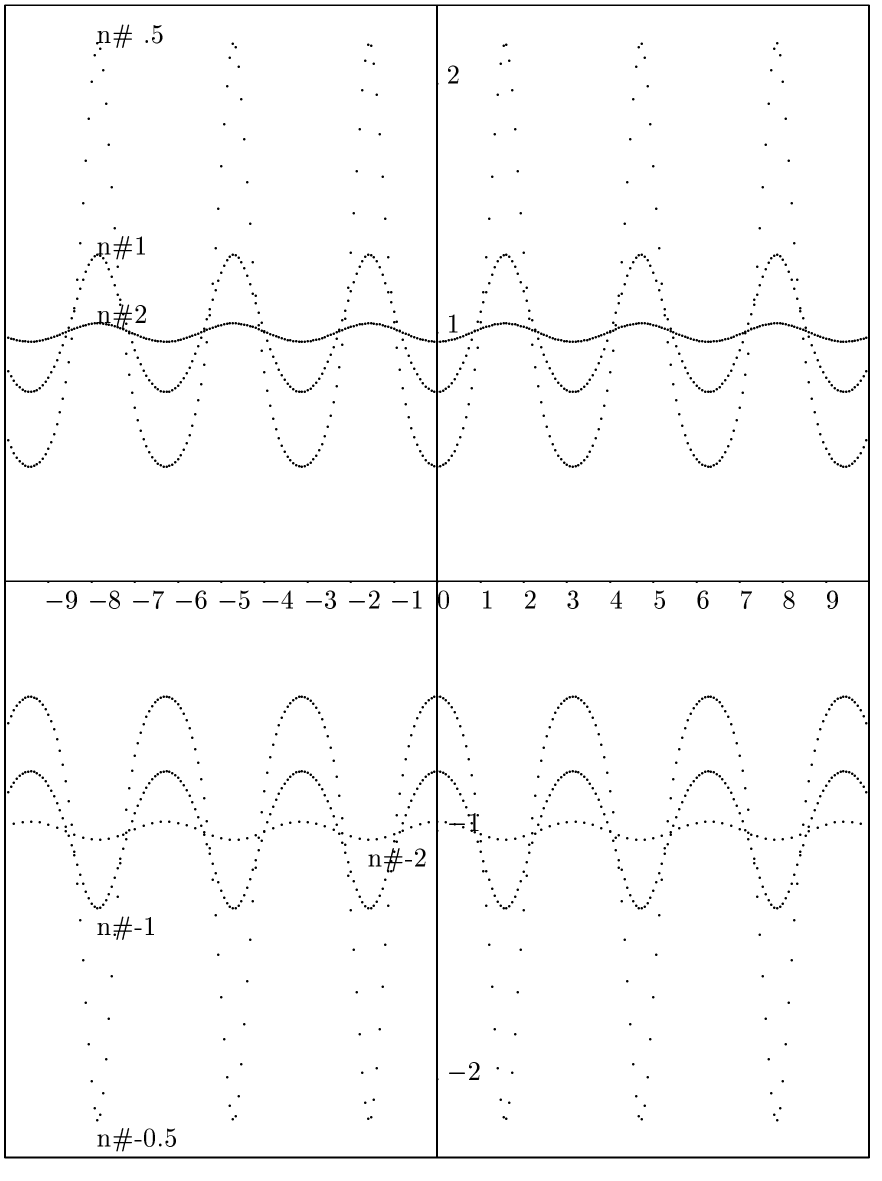


Figure P.11: Graph of y#9O5Ox+nX0j1

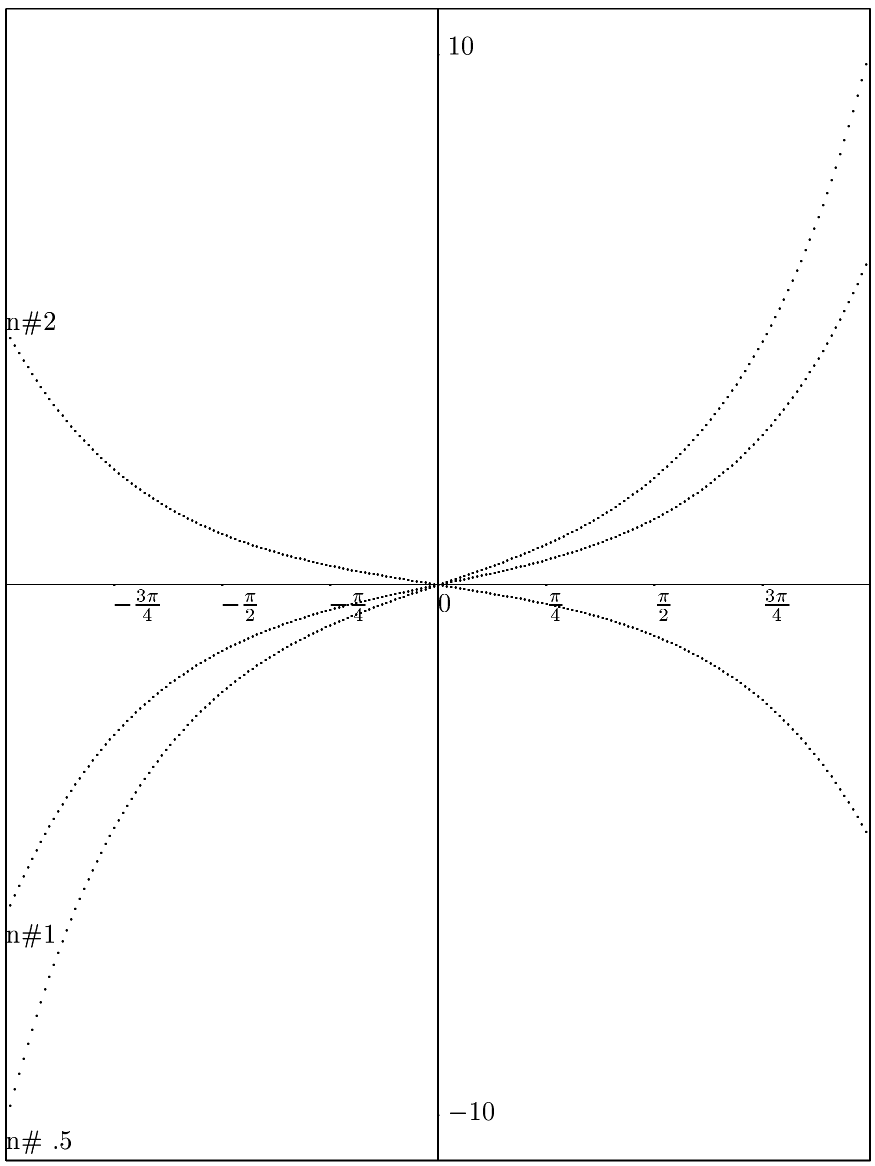


Figure P.12: Graph of y#11O5Ox+nX0j1

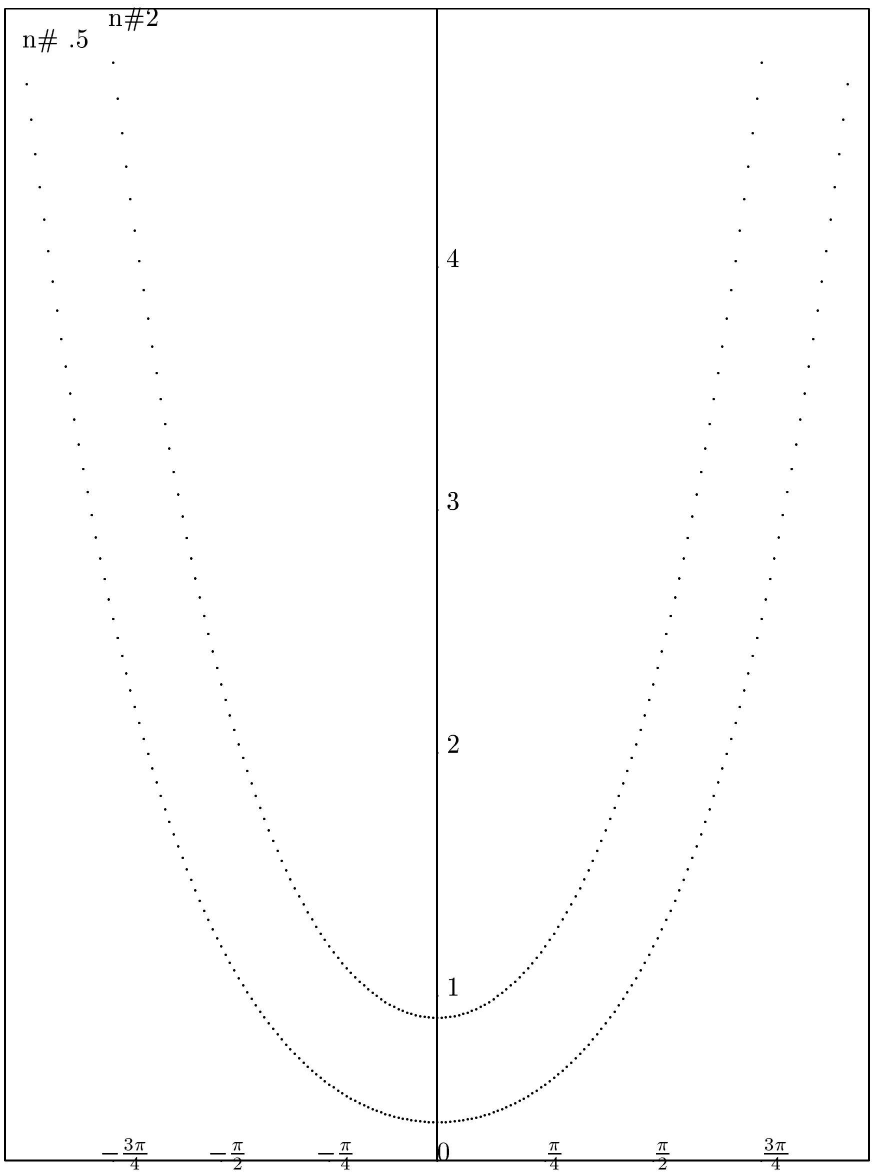


Figure P.13: Graph of y#9O5O .5+xX0j1

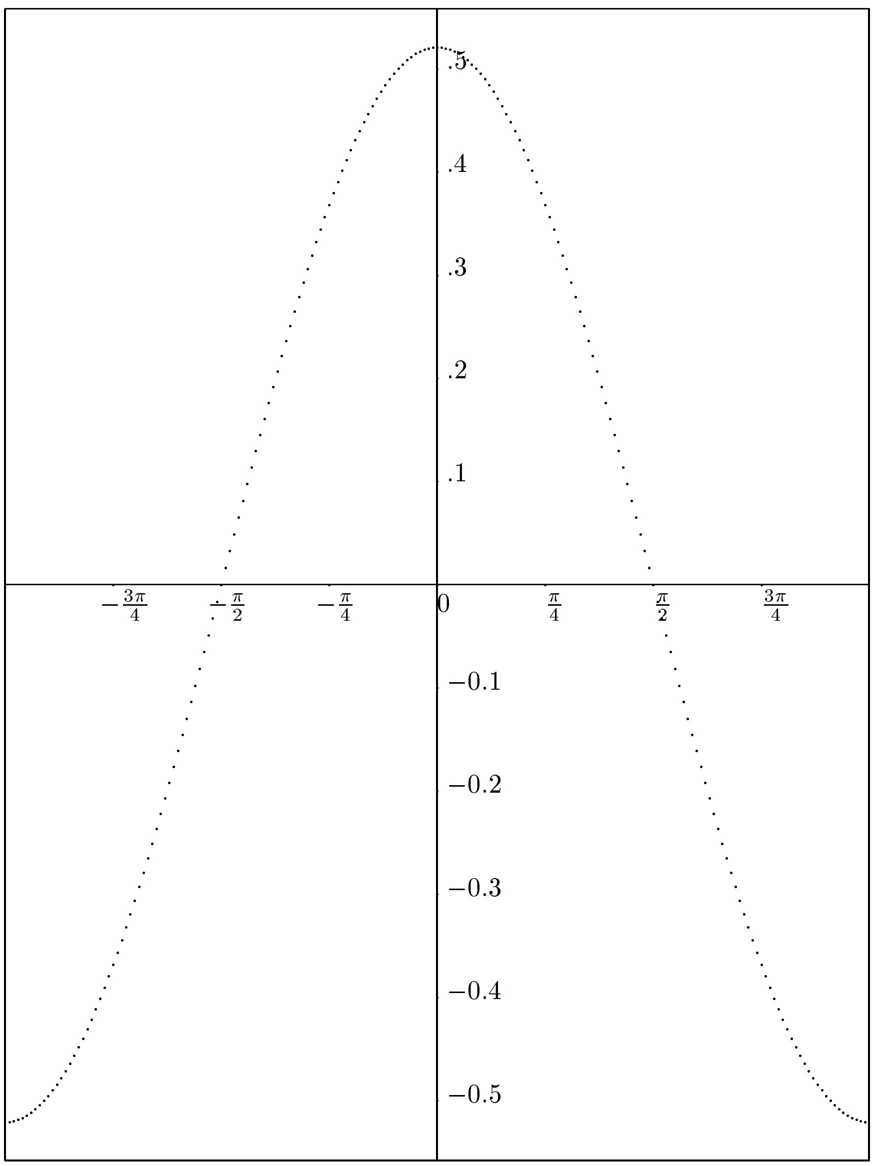


Figure P.14: Graph of y#11O5O1+xX0j1

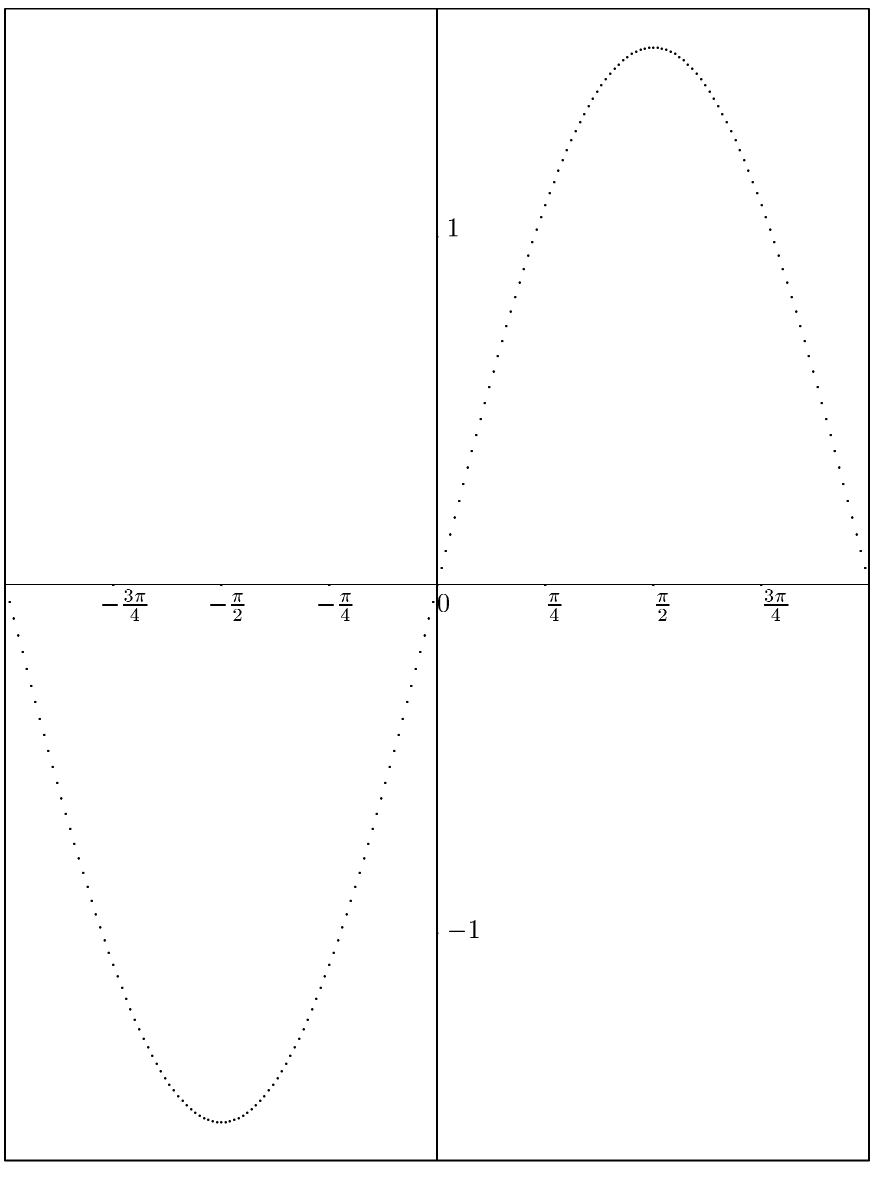


Figure P.15: Graph of y#9O6Ox+nX0j1

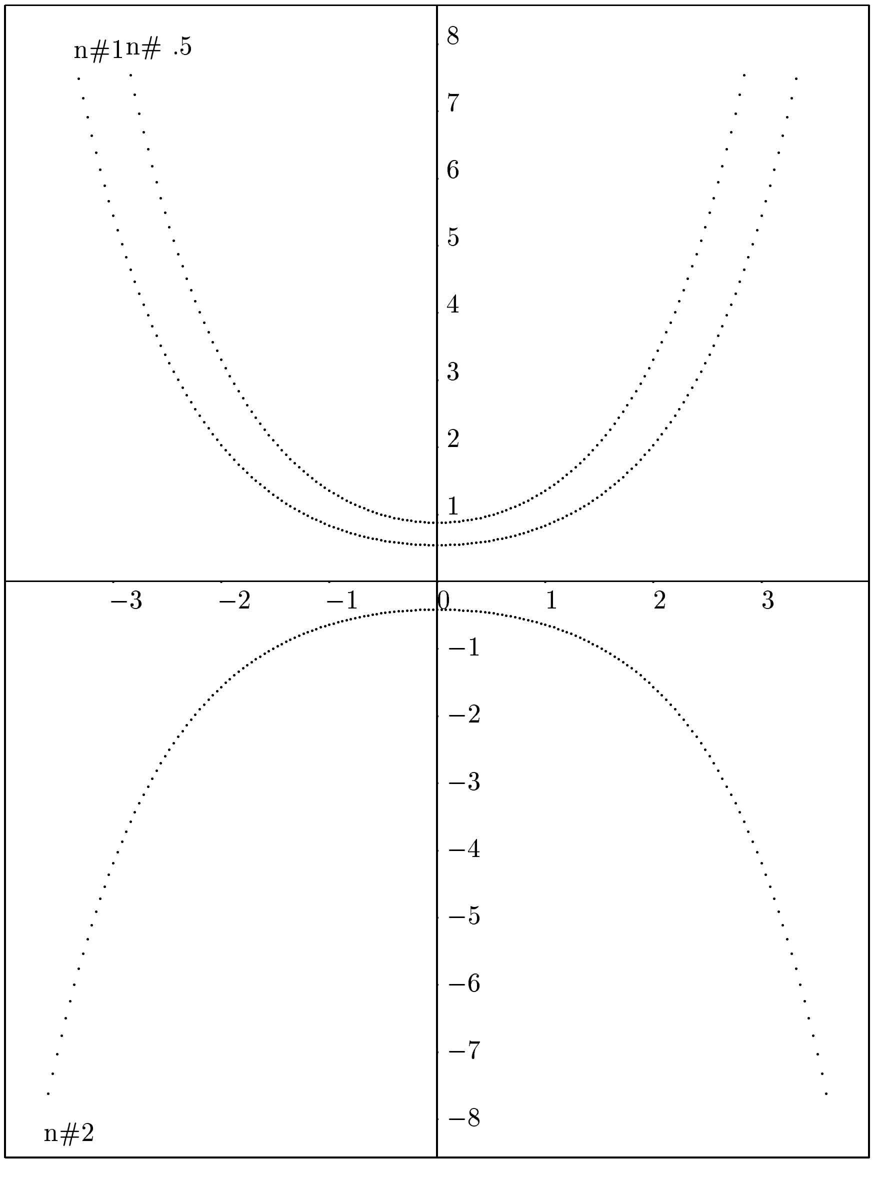


Figure P.16: Graph of y#11O6Ox+nX0j1

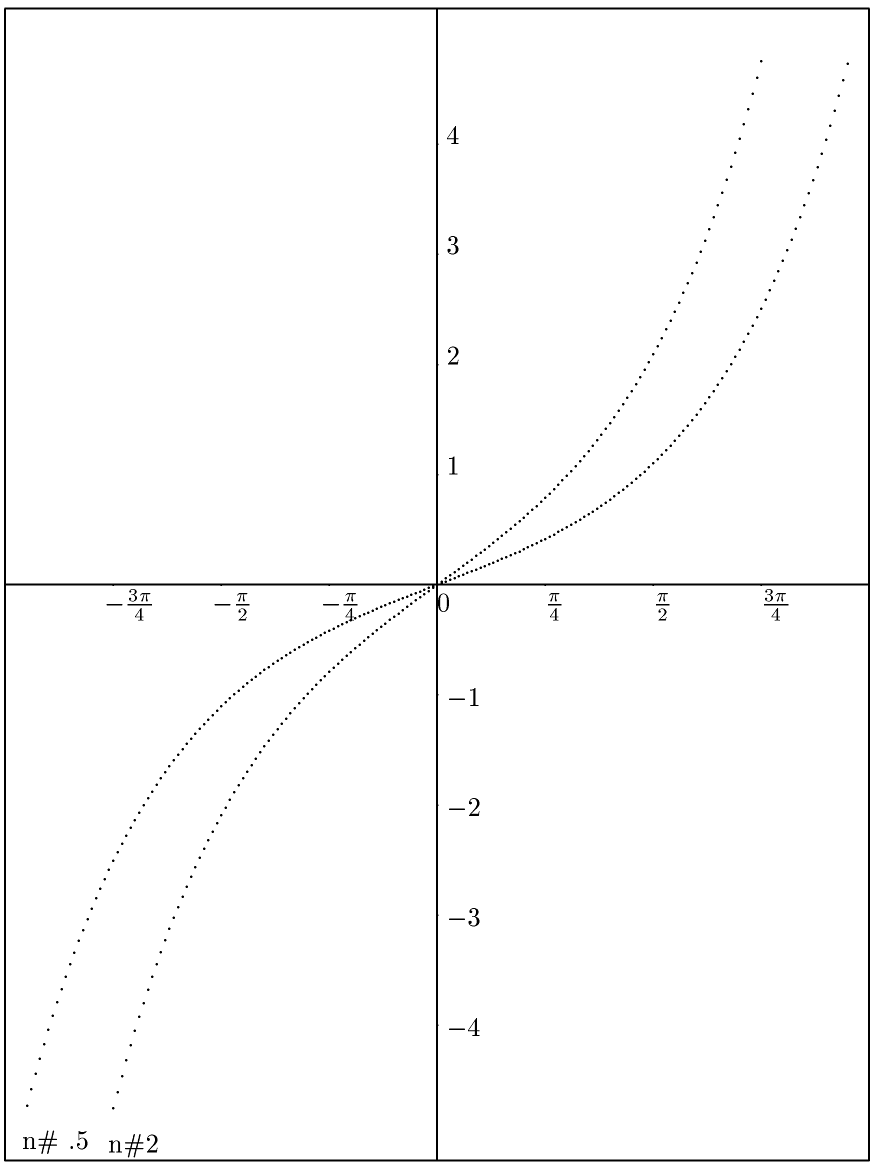


Figure P.17: Graph of y#9O6O .5+xX0j1

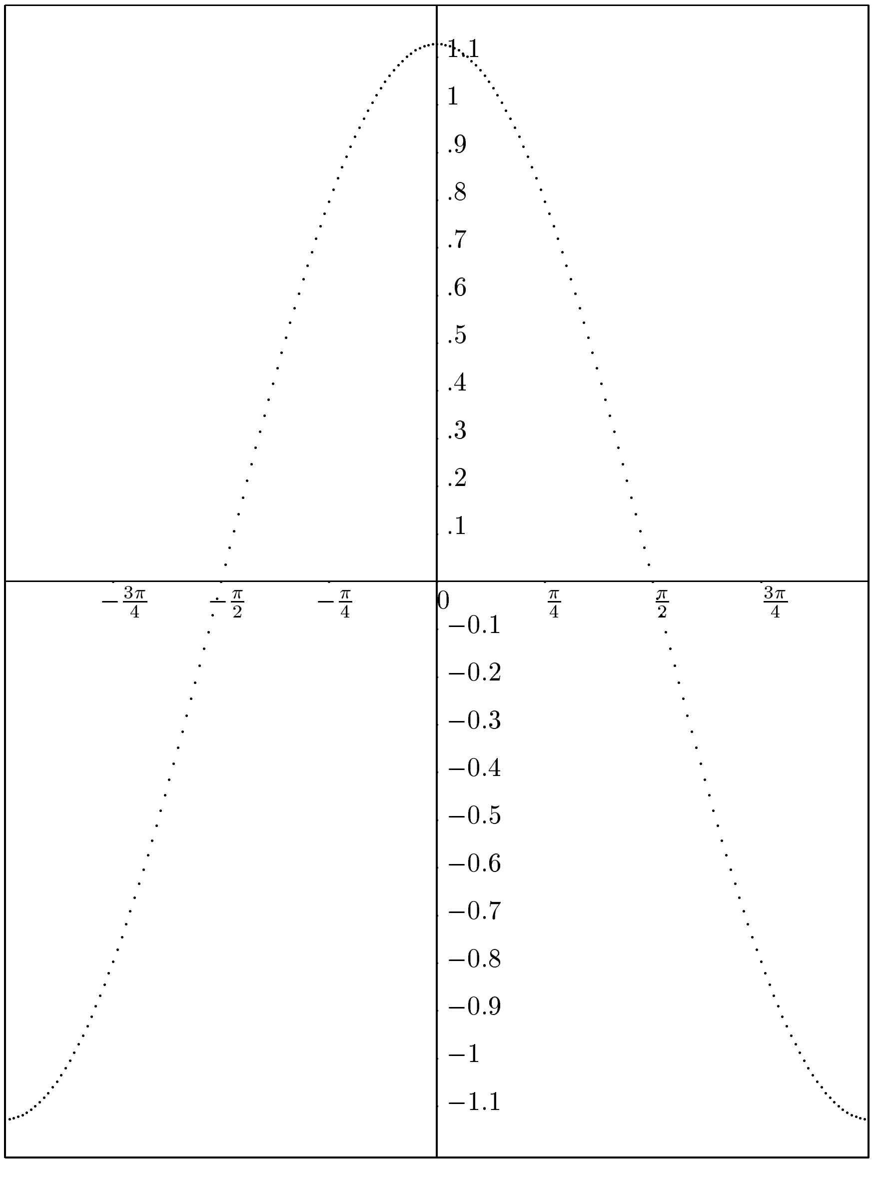


Figure P.18: Graph of y#11O6O1+xX0j1

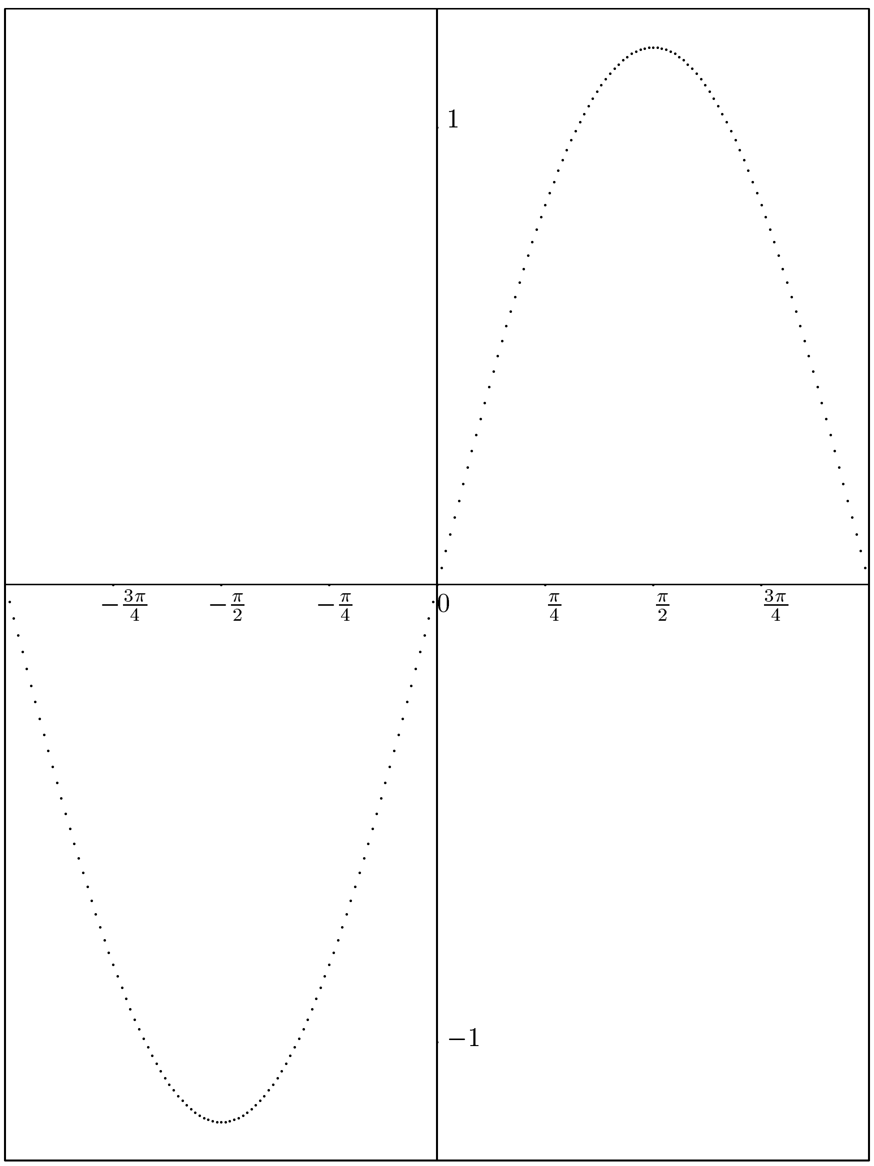


Figure P.19: Graph of y#7Ox

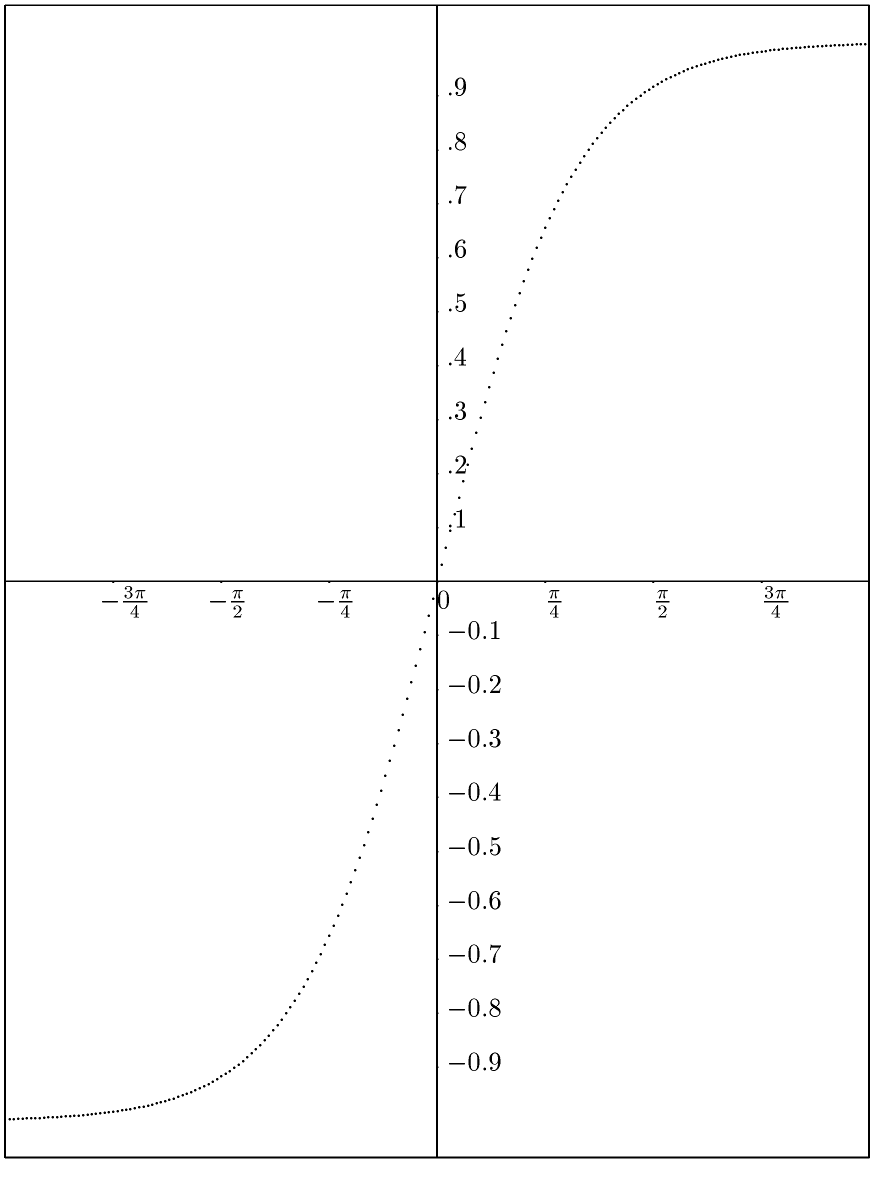


Figure P.20: Graph of y#9O7Ox+nX0j1

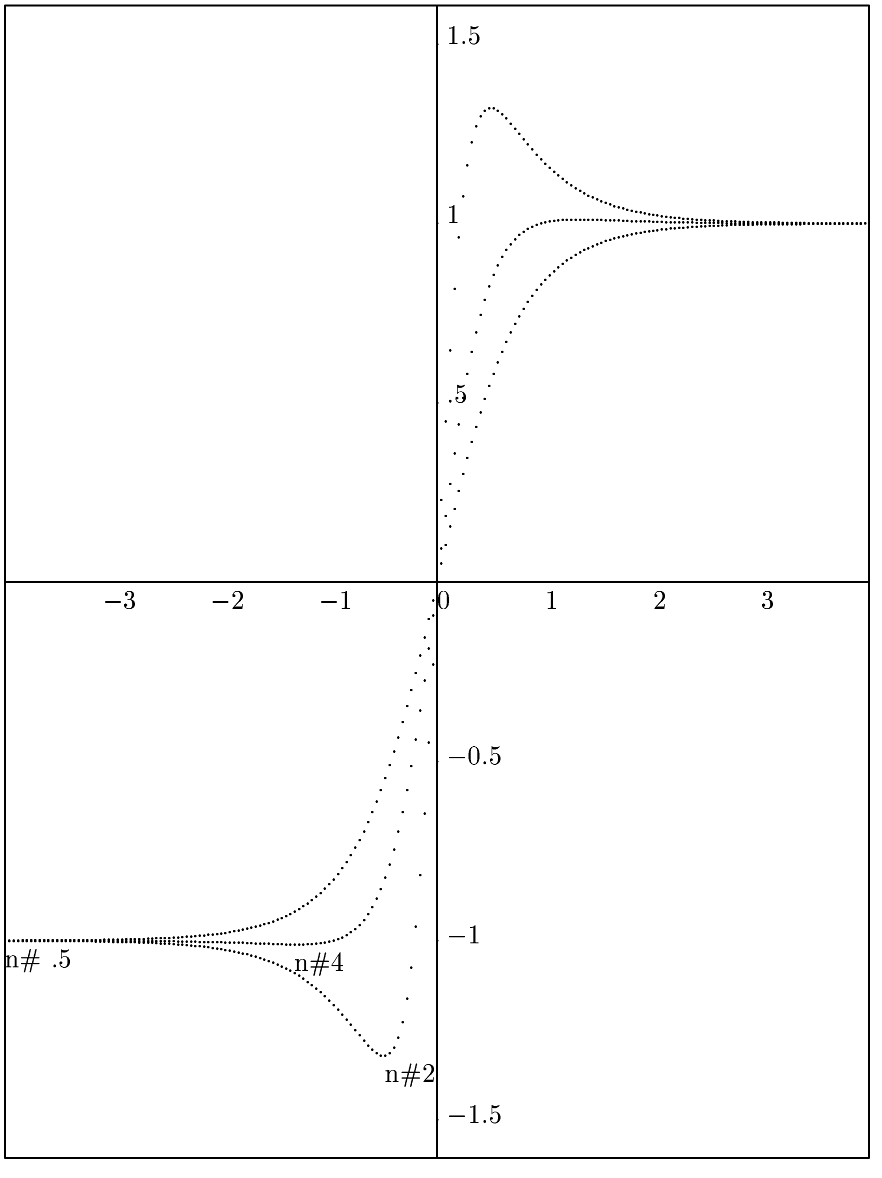


Figure P.21: Graph of y#11O7Ox+nX0j1

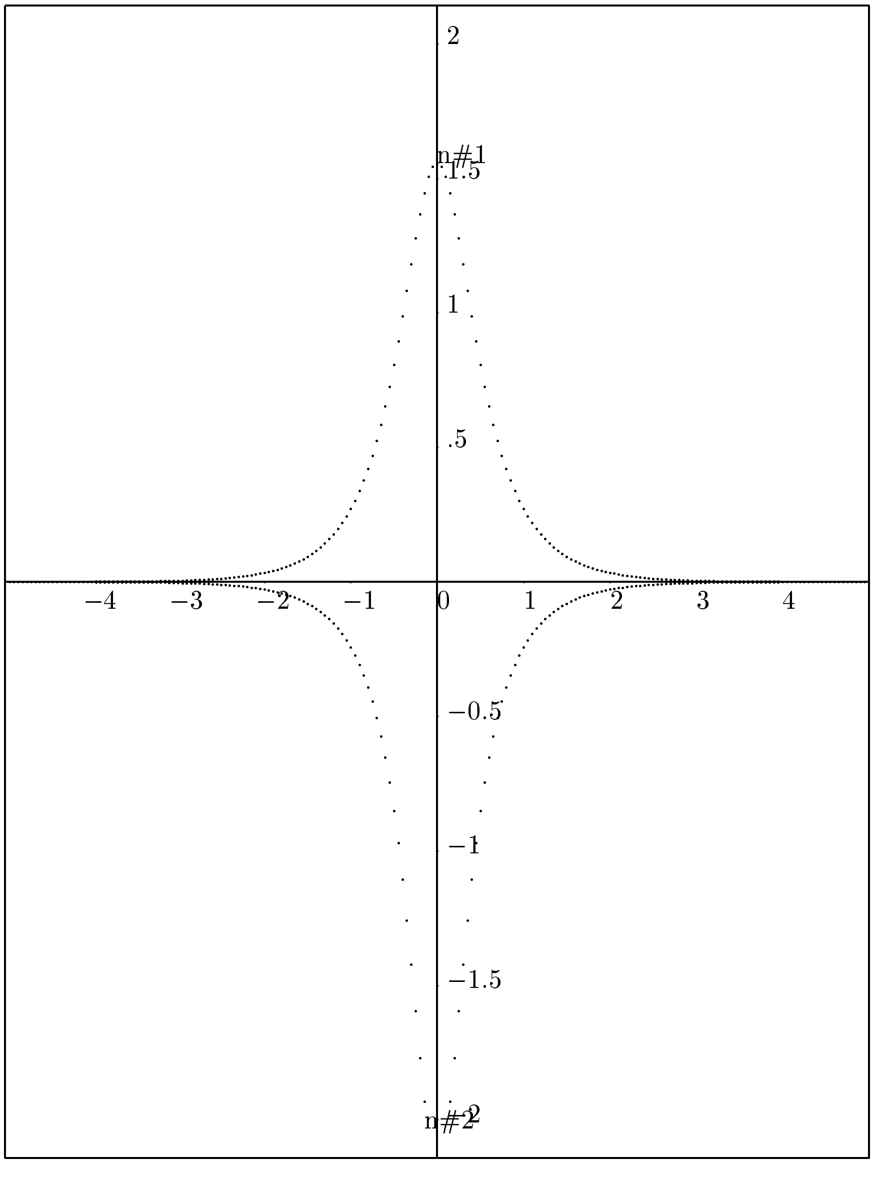


Figure P.22: Graph of y#9O7O .5+xX0j1

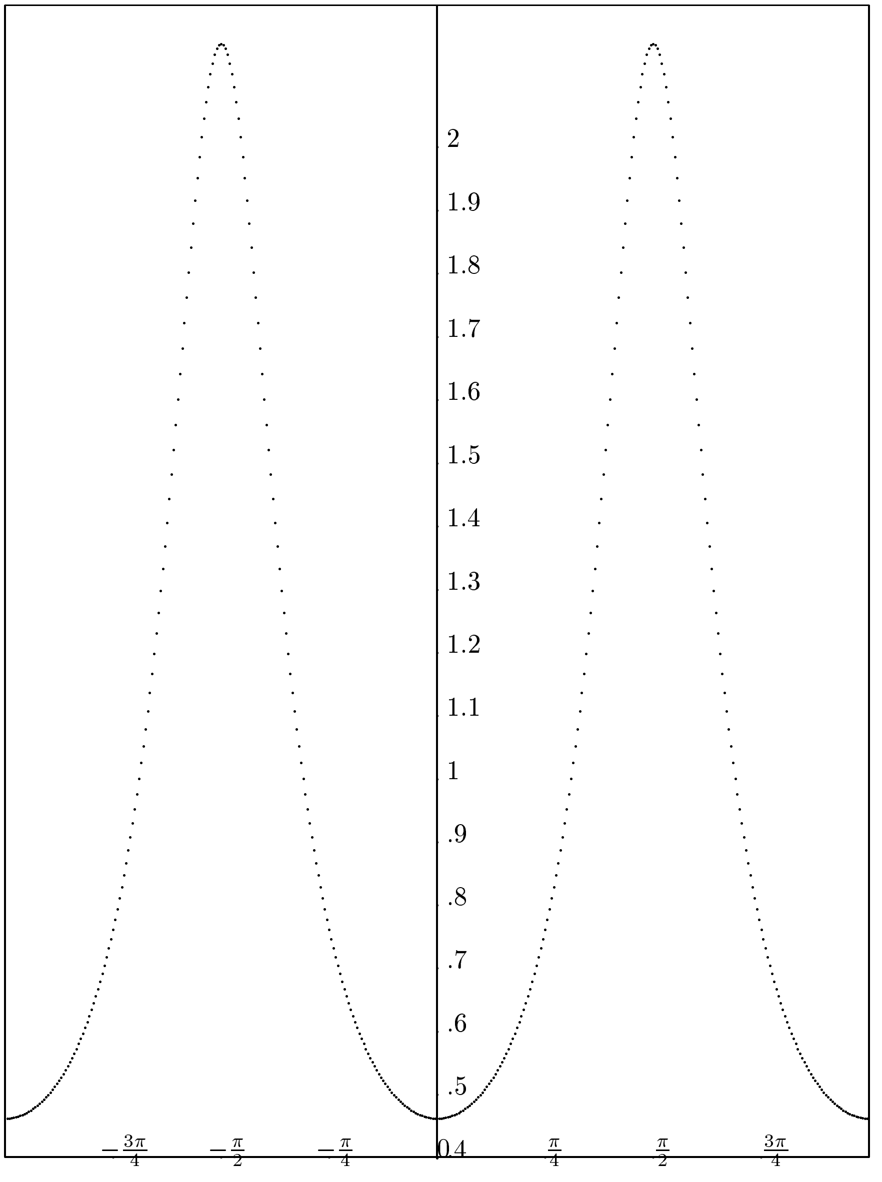


Figure P.23: Graph of y#11O7O1+xX0j1

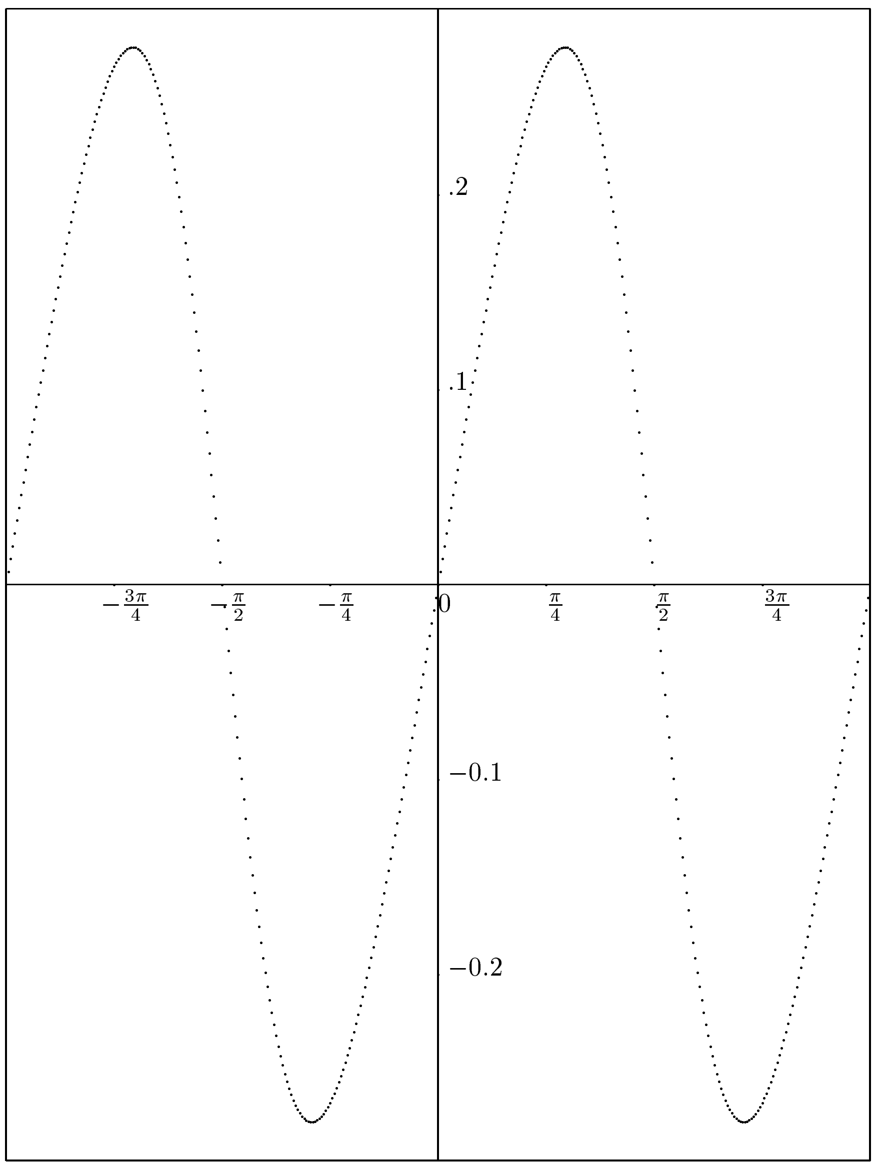


Figure P.24: Graph of y#*x

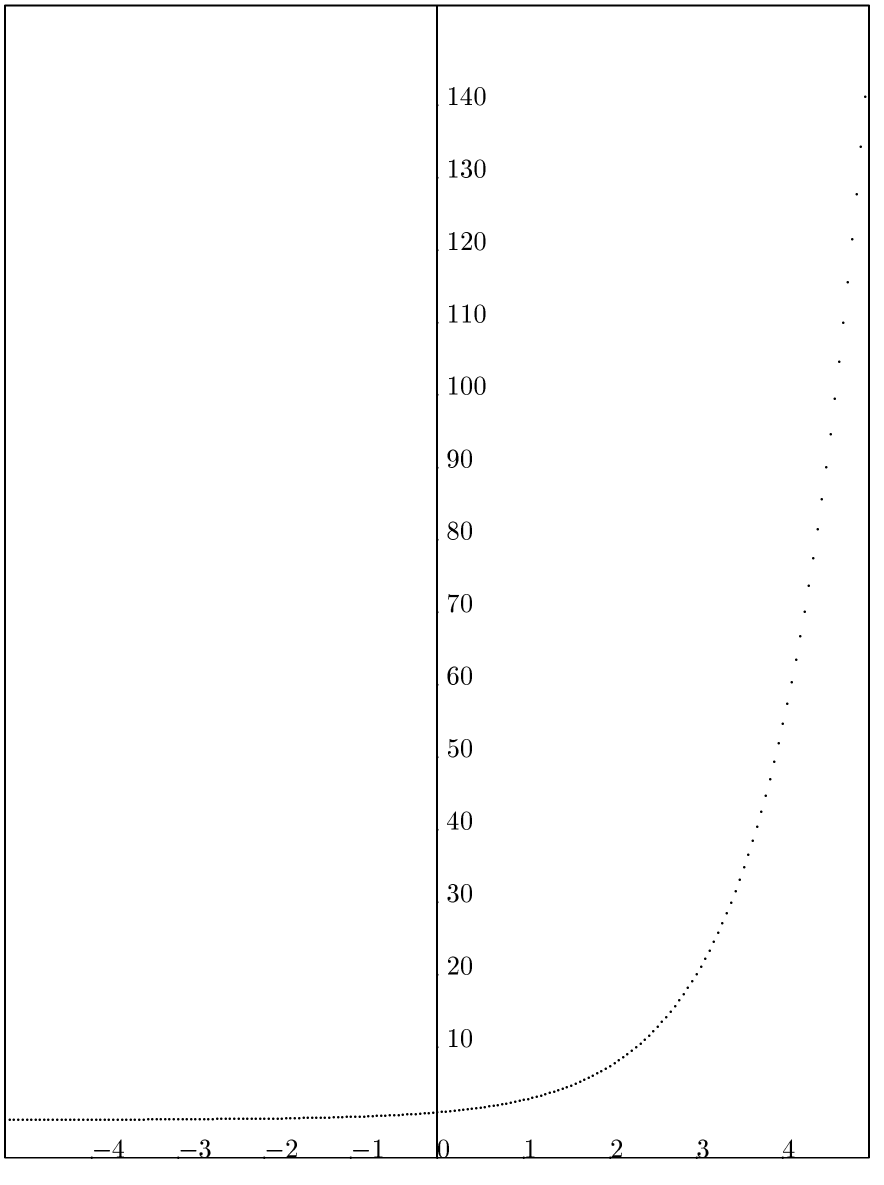


Figure P.25: Graph of y#9O*x+(n#1)X0j1

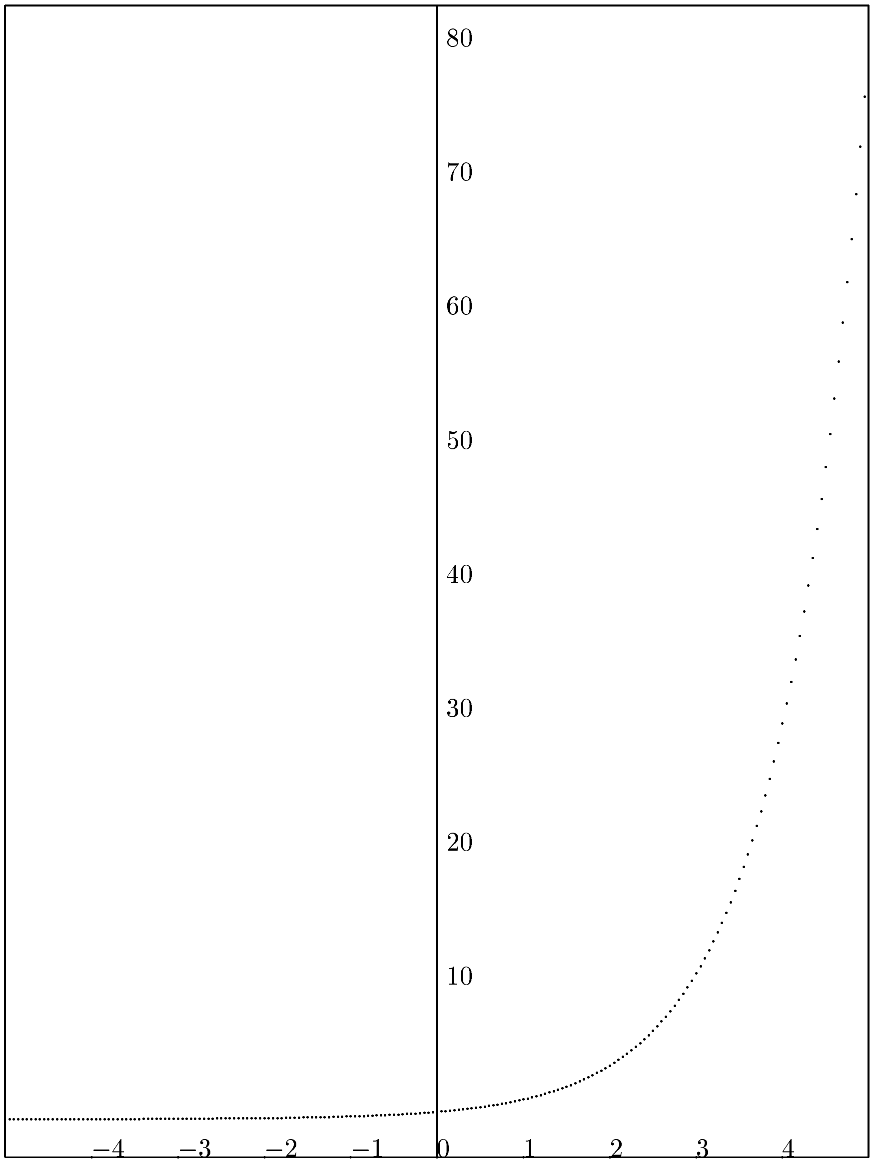


Figure P.26: Graph of y#11O*x+(n#1)X0j1

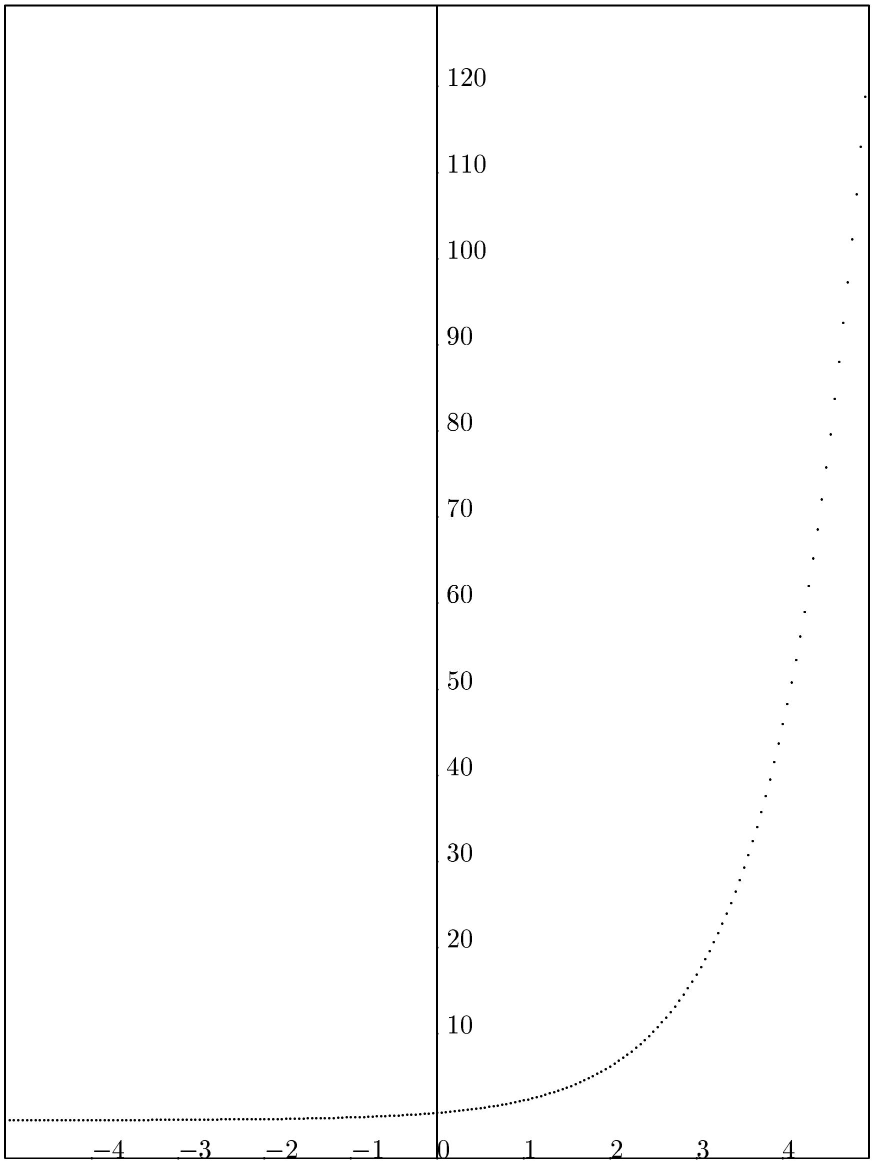
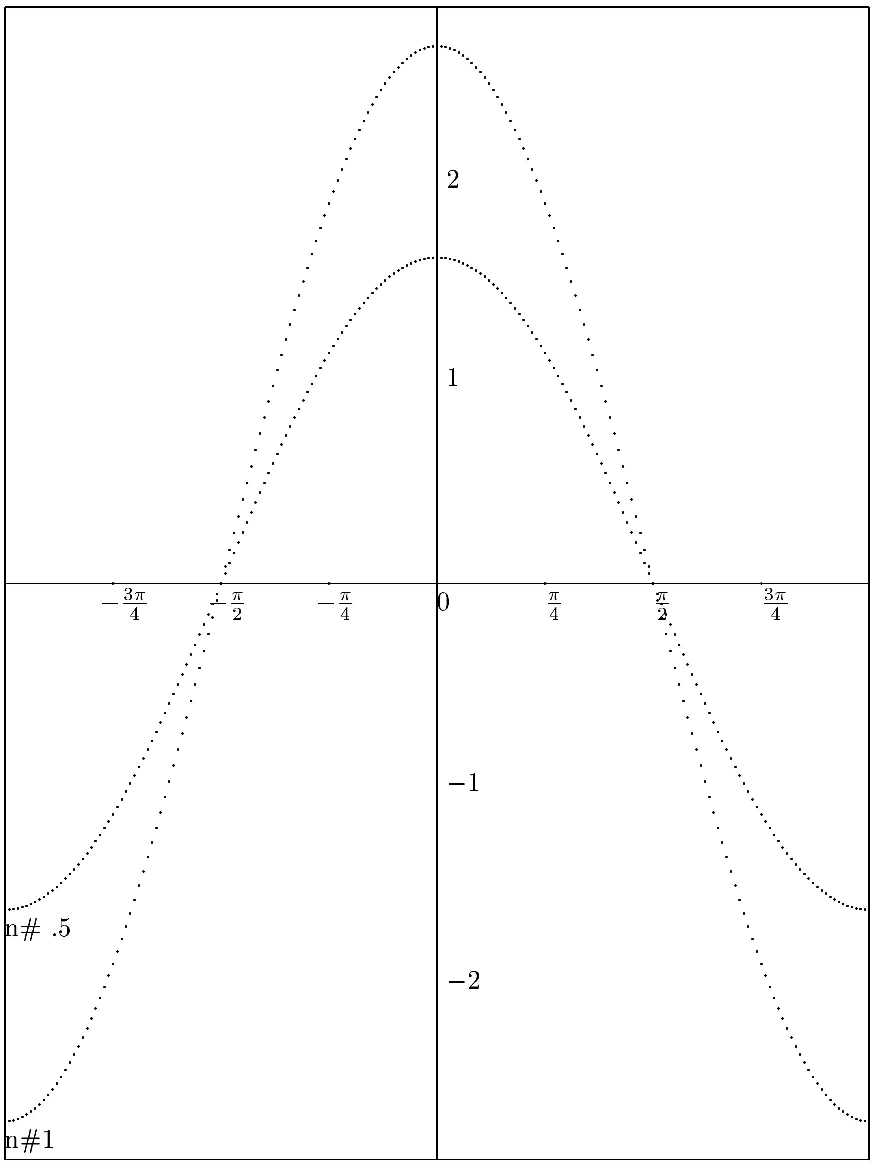


Figure P.27: Graph of y#9O*x+nX0j1



The data for all the graphs in this section were generated using CAPLIB2 statements. For example, data for Figure P.19 on page 454 were produced from this output of these statements:

```
)run \my\cap2\sample\graph.inc
"    htl: 0=both, 1=headers, 2=trailers, 3=neither.
"    | nlb 1 = Label the curve.
"    | | xr = 1=vary 90x, 0=vary 110x.
"    | | | e = i(11) or r(9) to select coeff. to graph.
"    | | | | yabm = maximum |y printed on graph.
"    | | | | | minx = minimum value of x.
"    | | | | | |maxx = maximum value of x.
"    | | | | | ||xiv = x axis marker interval.
"    | | | | | ||| hk = Constant coefficient of input.
"    | | | | | ||| | yiv = y axis marker interval, or 0.
"    | | | | | ||| | |   yca, 1 = ylwc, maxy, miny preset.
"    | | | | | ||| | |   |
"    V V V V V VVV V V   V
args#0,0,1,r,1,xpi,0,0.1,0
'70x' graph args
```

Here's a listing of `\my\cap2\sample\graph.inc`:

```
" Legend describing various global values:
"
" World coordinates(wc) are those of the real data.
" Graph coordinates(gc) are those of the graph.
"
" caption - Override to text for graph caption.
"           If null, a caption will be generated.
"           The graph function resets the global
"           caption variable to null at the end of
"           its processing.
"
" hk ------ Constant coefficient of input.  If xr=1
"           (see below) then hk becomes the constant
"           imaginary coefficient for all values of
"           x on the graph.  If xr=0, hk will be the
"           constant real coefficient.
"
" htl ----- 0 = both, 1 = headers, 2 = trailers,
"           3 = neither.
"
```

```
" maxx ---- Maximum x axis value in world coordinates.
"
" maxy ---- Maximum y axis value in world coordinates.
"
" minx ---- Minimum x axis value in world coordinates.
"
" miny ---- Minimum y axis value in world coordinates.
"
" mgc ----- Vertical margin in graphic coordinates.
"
" n ------- Synonymous with hk (see above).  The x
"           values to which the function is applied to
"           obtain y values are derived by first creating
"           xwc as a vector of integers uniformly
"           distributed between minx and maxx inclusive.
"           Then, either 'x#(nX0j1)+xwc' or 'x#n+0j1Xxwc'
"           is evaluated.
"
" nlb ----- 1 = Label the curve with the n value.
"
" points -- Number of points to generate.
"
" xgc ----- Array of x values for data points in graph
"           coordinates.
"
" xiv ----- x axis marker interval in world coordinates.
"
" xlin ---- Width of graph in inches.
"
" xpg ----- Divide xwc by xpg to get xgc.
"
" xpi ----- Array of three values for minx, maxx, and xiv,
"           used when invoking the graph function and the
"           array of x values spans -pi to +pi.
"
" xr ------ 1=vary real x coefficient, 0=vary imaginary
"           coefficient, holding the other coefficient to
"           the constant hk (see above.).
"
" xt ------ Used in a variety of places to temporarily
"           generate graphics coordinates.
"
```

```
" xwc ----- Array of x values in world coordinates.
"
" yadj ---- Adjustment down to print text below a line.
"
" yabm ---- Maximum absolute value (|y) to appear on graph.
"
" ygc ----- Array of y values for data points in graph
"           coordinates.
"
" ylin ---- Height of graph in inches.
"
" ymgn ---- Margin in inches at top and bottom of y axis.
"
" ypg ----- Divide ywc by ypg to get ygc.
"
" yt ------ Used in a variety of places to temporarily
"           generate graphics coordinates.
"
" ywc ----- Array of y values for data points in world
"           coordinates.
"
" Set global values. ------------------------------------->
"
caption#'' " Empty caption causes one to be generated.
i#11 " Circle function code to extract imag. coeff.
points#200 " Number of data points to generate on graph.
r#9 " Circle function code to extract real coeff.
xlin#4.5 " Width of graph in inches.
"  minx = -3.14159....
"  |     maxx = 3.14159....
"  |     |     xiv
"  |     |     |
"  V     V     V
xpi#(O-1),(O1),0.25
ylin#6 " Height of graph in inches.
ymgn#.2 " Margin in inches at top and bottom of y axis.
"
" <------------------------------------------------------>
" Generates the LaTeX \put statements for the data points
" appearing on the graph.
"
Lex 'dodata'
```

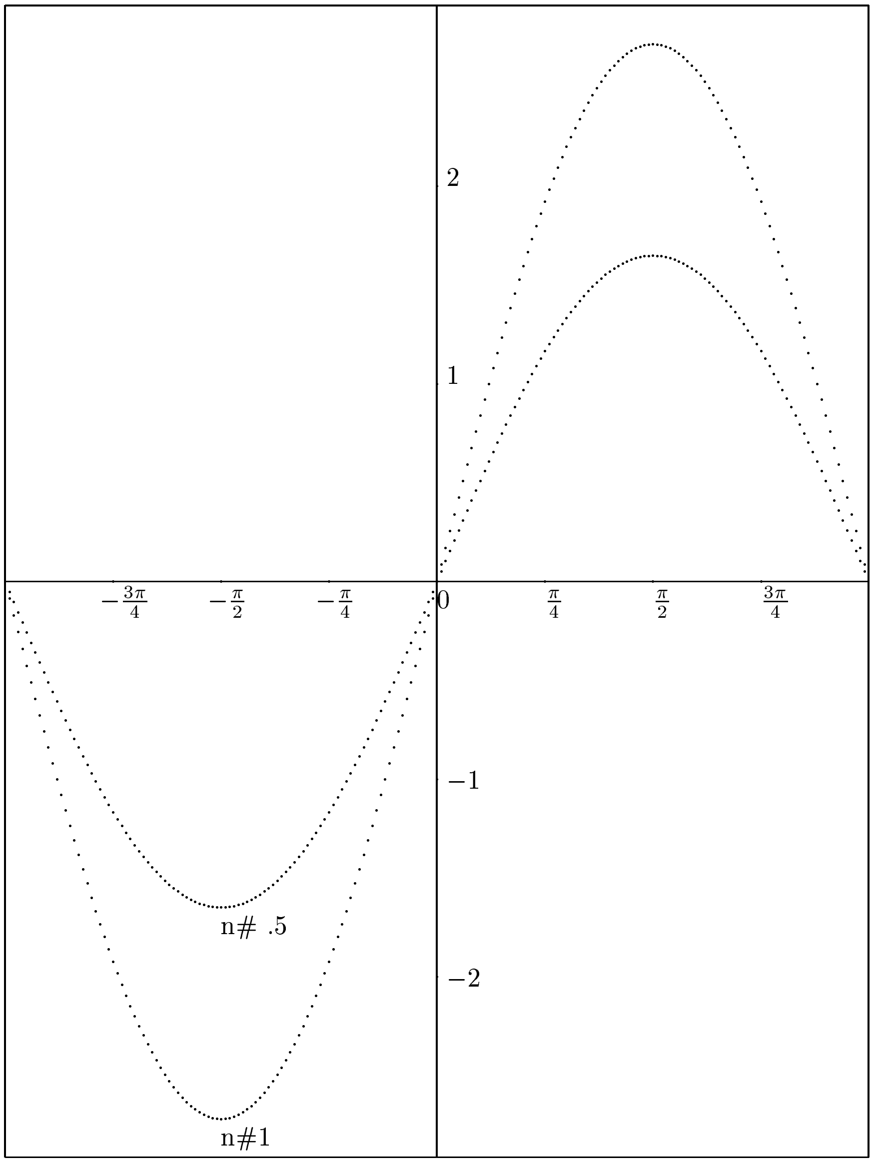
```
Gdodata
xgc#(xwc_minx)%xpg " xgc=x graphic coord. data points.
ygc#mgc+(ywc_miny)%ypg " ygc=y graphic coord. data points.
$bylabXI0=nlb " Branch if the curve is not to be labelled.
'%Label the curve'
xt#1Y(u=S/u#|ywc)/xgc " x coord where max/min occurs
yt#(_yadjX0>vs/ywc)+(vs#xt=xgc)/ygc " y coord of max/min
" Note: Calc. for yt works only if all minima occur below
" y axis, and all maxima occur above.
pcon,(xt,',',[1.5]yt),'Z'){n\#',(Fhk),'}'
bylab:'%Draw the data points'
pcon,((xgc#-1U1Uxgc),',',[1.5](ygc#-1U1Uygc)),circon
G
" <------------------------------------------------------->
" Generate xwc and ywc, the arrays of x/y coord. for the
" data points to appear on the graph.
"
Lex 'genxy'
Ggenxy
xwc#minx+(xlwc#maxx_minx)X(-1+Ipoints+1)%points
$varyrealXIxr
x#hk+0j1Xxwc " real part is constant, imaginary varies.
$calcy " Branch to compute values of y for data points.
varyreal:x#(hkX0j1)+xwc " Imag. is constant, real varies.
calcy:ywc#eOCfun " Compute values of y for data points
ywcm#yabm>|ywc " Mask of keepers, magnitudes of y < yabm.
xwc#ywcm/xwc " Pick the keepers.
ywc#ywcm/ywc " Pick the keepers.
G
"
" <------------------------------------------------------->
" Main graph routine.
"
Lex 'graph'
Gfun graph a
"Graphs the imaginary or real coefficient of result of fun.
" fun = expression to evaluate.
(htl nlb xr e yabm minx maxx xiv hk yiv yca)#a
genxy " Generate the data points.
$dataXIhtl>1 " Branch if htl greater than 1.
scale " Calculate global scaling values.
headers " Generate LaTeX figure headers.
```

```
data:dodata " Process and graph data points.
trailers " Generate Latex figure trailers, maybe.
G
"
" <------------------------------------------------------->
" Generates the LaTeX statements to begin the graph.
"
Lex 'headers'
Gheaders
'\begin{figure}[tbh]'
$gencapXI0=Rcaption " Branch if no caption override.
'\caption{',caption,'}'
$begin
gencap:$realcapXI(xr=1)&hk=0 " Branch if x not complex.
$ncaptionXInlb=0 " Branch if curves not labelled w/n val.
'\caption{Graph of y\#',(Fe),'O',fun,'+nX0j1}'
$begin
ncaption:$cplxcapXIxr " Branch if varying real coeff.
'\caption{Graph of y\#',(Fe),'O',(-1Ufun),(Fhk),'+xX0j1}'
$begin
cplxcap:"
'\caption{Graph of y\#',(Fe),'O',fun,'+(n\#',(Fhk),')X0j1}'
$begin
realcap:'\caption{Graph of y\#',fun,'}'
begin:'\begin{center}'
'\setlength{\unitlength}{',(Flin),'in}'
'\begin{picture}(',(Fxlin%lin),',',(Fylin%lin),')'
'%Draw a frame around the picture'
' \put(0,0){\line(1,0){',(Fxlgc),'}}% bottom'
' \put(0,0){\line(0,1){',(Fylgc),'}}% left'
' \put(0,',(Fylgc),'){\line(1,0){',(Fxlgc),'}}% top'
' \put(',(Fxlgc),',0){\line(0,1){',(Fylgc),'}}% right'
'%Draw the x axis'
' \put(0,',(Fxax),'){\line(1,0){',(Fxlgc),'}}%x axis'
xt#xoff%xpg
pcon,((xt,[1.5]',')',xax),circon " Draw the x axis markers.
xt#xt_xpgX.1Xxmk<0
yt#xax+((.05%lin)Xxax=mgc)_yadjXxax>mgc
$dopaxXIpix
'%Draw the x axis marker values'
pcon,xt,',',yt,econ,xmk,[1.5]scon
$doyax
```

```
dopax:'%Draw the x axis marker values in pi'
picon#(`Z'\frac{') ,`1 '\pi}{4}' '\pi}{2}' '3\pi}{4}'
picon#('-',`1`Rpicon),'0',picon
pcon,xt,',',yt,econ,picon,[1.5]scon
doyax:'%Draw the y axis'
$putymkXI(yax=0)
' \put(',(Fyax),',0){\line(0,1){',(Fylgc),'}}%y axis'
putymk:'%Draw the y axis markers'
ymask#ymk^=0
yt#ymask/mgc+(ymk_miny)%ypg
pcon,yax,',',yt,[1.5]circon
'%Draw the y axis marker values'
xt#yax+.05%lin
yt#yt_ypgX.1X(ymask/ymk)<0
pcon,xt,',',yt,econ,(ymask/ymk),[1.5]scon
G
"
" <------------------------------------------------------->
" Calculates many values needed to produce the graph.
"
Lex 'scale'
Gscale
$byyXIyca " Branch if ylwc, maxy, miny are precalculated.
ylwc#(maxy#S/ywc)_miny#D/ywc
byy:ylap#ylin_2Xymgn " ylap=height allowed data points.
lin#(xlin%xlwc)Dylap%ylwc " unitlength in inches.
yadj#.14%lin " y graphic coord. adj. to print below line.
mgc#ymgn%lin " Margin in graph coordinates.
xpg#xlwc%xlgc#xlin%lin " Divide xwc by xpg to get gc.
ypg#ylwc%(_2Xymgn%lin)+ylgc#ylin%lin
" Divide ywc by ypg to get gc.
xax#(yz#(minyK0)&maxyZ0)Xmgc+(|miny)%ypg
" xax is xaxis in graph coord.
yax#(xz#(minx<0)&maxx>0)X(|minx)%xpg
" yax is yaxis in graph coordinates.
$piaxisXIpix#(minx=O-1)&maxx=O1
" branch if pi units on x axis.
xic#(yax=0)+Dxlwc%xiv
$doyiv
piaxis:xic#Dxlwc%xiv#O.25
doyiv:$doyicXIyiv^=0
yiv#10*D10@ylwc
```

```
doyic:yic#yic+0=2|yic#Dylwc%yiv
xoff#(I-1+xic)Xxiv
" xoff is offset from minx in world coord. of x markers.
yoff#(_yiv)+(Iyic)Xyiv
" yoff is offset from miny in world coord. of y markers.
$yoffplusXIminy>0
ymk#yoff+miny+yiv||miny
$yoffdone
yoffplus:ymk#yoff+miny_yiv|miny
" ymk is y for y axis markers in world coord.
yoffdone:xmk#minx+xoff
" xmk is x for x axis markers in world coord.
circon#`Z'){\circle*{',(F.0205%lin),'}}'
scon#`Z'$}'
econ#`Z'){$'
pcon#`Z' \put('
G
"
" <------------------------------------------------------->
" Generates the LaTeX statements to finish the graph.
"
Lex 'trailers'
Gtrailers
$epicXIhtl=0 " Branch if both headers and trailers.
$eojckXInlb " Branch if graph already labelled.
pcon,(1Yxgc+xpgX.1),',',(1Yygc),'){',fun,'}' " Label graph.
eojck:$eojXI(htl=1)+htl=3 " br if headers only, or neither.
epic:'\end{picture}'
'\end{center}'
eoj:'%Finis.'
caption#'' " Reset global caption
G
```

Figure P.28: Graph of y#11O*x+nX0j1



Appendix Q

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Q.1 Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

Q.2 GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.

2. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

a) cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and

b) cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.

d) You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

3. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

a) accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,

b) accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,

c) accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

4. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.

5. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

7. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.

8. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the

two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

9. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

10. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

- <one line to give the program's name and a brief idea of what it does.> Copyright (C) 19yy <name of author>

- This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 1, or (at your option) any later version.

- This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

- You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19xx name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here's a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (a program to direct compilers to make passes at assemblers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

That's all there is to it!

Bibliography

[1] IBM, *APL Language*, GC26-3847-4, White Plains, New York, 1978.

[2] IBM, *The APL Handbook of Techniques*, White Plains, New York, 1978.

[3] IBM, *APL2 Programming: Lanugage Reference, Version 2 Release 1*, SH21-1061-00, San Jose, California, 1992.

[4] *Handbook of Mathematical Tables and Formulas*, compiled by Richard Stevens Burington, Ph. D., Sandusky, Ohio, Copyright 1933, 1940, 1948.

Index