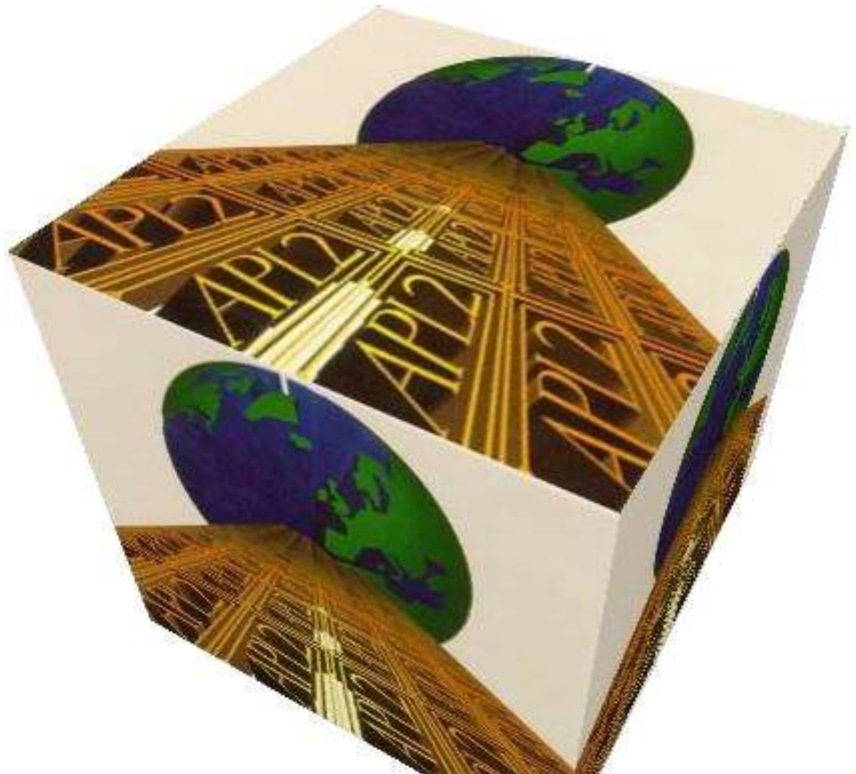
APL2 Language Summary

SX26-3851-02

APL Products and Services
IBM Silicon Valley Laboratory
555 Bailey Avenue
San Jose, California 95141
APL2@vnet.ibm.com



Copyrights

Contents

- Notices
- We Would Like to Hear from You
- Introducing APL2
- Primitive Functions
- Primitive Operators
- System Functions
- System Variables
- System Commands
- Defined Functions and Operators
- Error Reports and Error Codes
- Glossary

Notices

Programming Interface Information

This language summary is intended to help programmers write applications in APL2. It documents *General-Use Programming Interface and Associated Guidance Information* provided by APL2. General-use programming interfaces allow the customer to write programs that obtain the services of APL2.

Trademarks

We Would Like to Hear from You

APL2 Language Summary

Please let us know how you feel about this online documentation by placing a check mark in one of the columns following each question below:

To return this form, print it, write your comments, and mail it to:

International Business Machines Corporation
APL Products and Services - PGUA/E1
555 Bailey Avenue
San Jose, California 95141
USA

For postage-paid mailing, please give the form to your IBM representative.

You can also send us your comments by email. To send us this form, copy it to a file, write your comments using a file editor, and then send it to:

apl2@vnet.ibm.com

Overall, how satisfied are you with the online documentation?

```
                          Very               Very
                          Satisfied   Dissatisfied
                           1      2      3      4
  Overall Satisfaction    ___    ___    ___    ___
```

Are you satisfied that the online documentation is:

```
  Accurate                ___    ___    ___    ___
  Complete                ___    ___    ___    ___
  Easy to find            ___    ___    ___    ___
  Easy to understand      ___    ___    ___    ___
  Well organized          ___    ___    ___    ___
  Applicable to your tasks ___   ___    ___    ___
```

Please tell us how we can improve the online documentation:

```
  ________________________________________________________________
  ________________________________________________________________
  ________________________________________________________________
```

Thank you! May we contact you to discuss your responses?

```
  ___Yes  ___No
  Name:
```

```
______________________________________________
Title:
______________________________________________
Company or Organization:
______________________________________________
Address:
______________________________________________
______________________________________________
______________________________________________
Phone:
(___)_________________________________________
E-mail:
______________________________________________
```

Please do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office servicing your locality.

Introducing APL2

If you're new to APL, this material may provide some understanding of what it is all about; we'll discuss the syntax and characteristics of the language.

If you're *not* new to APL, we *still* recommend looking through what follows. APL2 offers capabilities that are not present in other versions of APL. Reading this may acquaint you with some features you are not familiar with, particularly if the versions you have used have not included APL2-style nested arrays.

Here are the major sections of this introduction:

- APL - What Is It?
- Getting Started in APL
- Fundamentals

APL - What Is It?

APL2 is a general-purpose language that enjoys extensive use in such diverse applications as commercial data processing, system design, mathematical and scientific computation, and the teaching of mathematics and other subjects. It has proved to be particularly useful in data base applications, where its computational power and communication facilities combine to enhance the productivity of both application programmers and end users.

When implemented as a computing system, APL2 is used from a typewriter-like keyboard. Statements that specify the work to be done are entered by typing them, and in response, the computer displays the result of the computation. The result appears at a device that accompanies the keyboard, such as video display or printer. In addition to work that is performed purely at the keyboard and its associated display, entries may also invoke the use of printers, disk files, or other remote devices.

The letters "APL" originated with the initials of a book written by Dr. Kenneth E. Iverson, ***A P**rogramming* ***L**anguage* (New York: Wiley, 1962). Dr. Iverson developed the APL language first at Harvard, and then at IBM, in collaboration with Adin Falkoff and others. The term ***APL*** now refers to the language that is an outgrowth of that work.

APL2 is a particular implementation of that language including array extensions researched by Trenchard More, elaborated by James A. Brown in his doctoral dissertation, and developed at IBM under the direction of Dr. Brown.

For additional highlights of the language, see:

- Power, Relevance, and Simplicity
- A Short Example of the Use of APL
- The Characteristics of APL

Power, Relevance, and Simplicity

A programming language should be relevant. That is, you should have to write only what is logically necessary to specify the job you want done. This may seem an obvious point, but many programming languages force you to be concerned as much with the internal requirements of the machine as with your own statement of your problem. APL2 takes care of those internal considerations automatically.

A programming language needs both power and simplicity: The power to handle large or complicated tasks, and the simplicity to state what must be done briefly and neatly, in a way that is easy to read and easy to write. You might think that power and simplicity are competing requirements, but that is not necessarily so. The power of APL as a programming language comes in part from its simplicity; it is this simplicity that makes it simultaneously well suited to the beginner and to the advanced user.

A Short Example of the Use of APL

Problems can often be solved in APL without writing programs, or even dealing with named variables. Simply typing in the expression to be evaluated causes the result to be displayed. Let's try out an example:

> Many bacteria can duplicate themselves once every half hour. If a single infectious organism began reproducing at 9 o'clock in the morning, how fast would the resultant colony grow?

Let's compute the number of bacteria at noon, 6 PM, and midnight. In other words, 3 hours, 9 hours, and 15 hours later. It's a doubling process, so we want 2 to some power. We write this as

```
      2⋆ ...
```

where `...` is not APL, it just indicates we haven't finished yet. Since the doubling happens twice every hour, we continue like this

```
      2⋆2× ...
```

Show the hours we are interested in, and we have told the system all it needs to know to solve the problem:

```
      2⋆2×3 9 15
64 262144 1073741824
```

So by noon there are 64 bacteria, but by 6 PM there are more than a quarter of a million, and by midnight there are over a *billion*!

Several distinctive features of APL2 are illustrated in this example: familiar symbols, such as × for multiplication, are used where possible, other symbols are introduced where necessary to make the notation linear (such as the ⋆ instead of a raised expression for the power function), and (***very*** important!) *a group of numbers can be worked on together*.

The Characteristics of APL

The primitive objects of the language are arrays (lists, tables, lists of tables, and so forth). For example, `A+B` is meaningful for any arrays `A` and `B`.

The syntax is simple. There is no hierarchy of function precedence, and built-in functions and user-defined functions (programs) are treated alike.

The rules of *programming grammar* are few. The definitions of the built-in functions are independent of the type of data to which they apply, and they have no hidden side effects.

The sequence control is simple. One statement type embraces all types of non-sequential flow (return, conditional execution, case, goto), and the termination of the execution of any function always returns control to the point of use.

External communication is established by means of data that is directly shared between APL and other systems or subsystems. These *shared variables* are treated both syntactically and semantically like other data. A subclass, called *system variables*, provides convenient communication between APL programs and their environment.

The utility of the built-in functions, called *primitive functions*, is vastly enhanced by *operators* which modify their behavior in a systematic manner. For example,

- *Reduction* (denoted by `/`) modifies a function to apply over all elements of a list, as in `+/L` for summation of the elements of `L`.
- *Axis specification* (denoted by `[n]`) allows functions to be applied to a table in a specified direction.

In addition, APL2 allows you to define both functions and operators for your own needs, and such user-defined programs are syntactically equivalent to primitive functions and operators.

The number of primitive functions is small enough that each is represented by a single easily-read and easily-written symbol, yet the set of primitives embraces operations from simple addition to complex types of formatting and advanced mathematical concepts such as hyperbolic cosine and matrix inversion.

Getting Started in APL

Here are some topics that can help you get started using APL:

- APL Is Interactive
- Who Typed What?
- Expressions
- The APL Character Set

APL Is Interactive

The APL2 system takes one APL2 expression at a time, converts it to *machine instructions* (the computer's internal language), executes it, and then proceeds to the next line. This is in contrast to traditional program compilers which convert complete programs to machine language before executing any expressions. This allows you a high degree of interaction with the computer. If something that you enter is invalid, you will get quick feedback on the problem before you proceed further.

Who Typed What?

Typically APL2 conducts a long-running conversation between the user and the APL system. The conversation is recorded in a log, managed by an APL2 component called the *Session Manager*. When you start APL, the session manager opens any existing log, letting you review or reuse previous information in it. The period of time during which the log is open is called an APL2 session.

During an APL2 session, you and APL2 will take turns adding information to the log. While you type information in, APL2 waits for some signal from you that it is *its* turn to use the information and display the results in the log. The normal signal from you is pressing the **Enter** key. When you press **Enter**, the session manager changes a mode indicator from *Input* to *Running*. (This indicator is normally in the lower left corner of the log window.) When it is your turn again, the mode is changed back to *Input*.

The session manager mode indicator is very useful while you and APL2 are "talking" to each other. However it does not help when you go back later to review the conversation. If you have a color display, you can tell the APL2 session manager to display input and output in different colors, but even that is of no use when reviewing earlier APL2 sessions, since the log is just a standard operating system file recording the conversation.

When APL2 displays information for you, it starts each new line at the left margin. After it finishes displaying any such output, it signals to you that it is ready for you to type in another keyboard input by indenting six spaces from the left margin and halting. This position is another indication that it's ready for you to take "your turn", and is useful when rereading the log.

```
      2+2
4
      3×4
12
```

Expressions

A typical *expression* in APL2 is of the form:

```
      AREA←3×4
```
The effect of the statement is to assign to the *name* `AREA` the value that is the result of `3×4` to the right of the *assignment arrow* (←). The expression may be read informally as "AREA *is* three times four."

If the leftmost part of an expression is not a name followed by an assignment arrow, the result of the expression is displayed. For example:

```
      3×4
12
      PERIMETER←2×(3+4)
      PERIMETER
14
```

The *leftmost* part of the expression is significant here, because APL2's order of evaluation is right-to-left. The leftmost part of the line, therefore, is the last part to be evaluated. But we'll get to the order of evaluation rules a little later on.

In this documentation, `CAPITALS` in the `Courier APL2` font are used to indicate the portions of the examples that you might actually see on your display, even though you can use lower case letters in an actual APL2 session, and can choose whether you wish to see italic or upright characters. Traditionally APL systems have used italic capitals, so you will see that convention used throughout the literature.

In addition to the **Enter** key, you can send other signals to APL2 by using the session manager's **Signals** menu. These signals are normally used while APL2 is running, rather than waiting for input from you. The **Attention** signal says "stop when it's convenient", and the **Interrupt** signal says "stop immediately".

The APL Character Set

The characters that may occur in a statement fall into four main classes: alphanumeric, operational, special, and blank.

- Alphanumeric characters are used in names and constants.
- Operational characters are used to represent operations (called functions) that are to be applied to data, or in some cases, represent what are called operators, that modify the way functions behave.
- Special characters include characters like parentheses that are not functions or operators, but affect the interpretation of the expression.
- The blank serves as a separator to mark divisions between names.

Fundamentals

The topics here provide a more complete overview of the language than the section on Getting Started in APL, but are still not a formal presentation. You will find complete details in the chapters on *Arrays* and *Syntax and Expressions* in *APL2 Programming: Language Reference*.

- Names
- Numbers
- Functions
- Operators
- Data
- Assigning Values to Names
- Order of Evaluation
- Errors

Summaries of facilities provided by specific language elements can be found under:

- Primitive Functions
- Primitive Operators
- System Functions
- System Variables
- System Commands

See also Defined Operations.

Again, the *Language Reference* provides complete information.

Names

Valid characters for forming names are:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ∆
a b c d e f g h i j k l m n o p q r s t u v w x y z ⍙
Ç ü é â ä à å ç ê ë è ï î ì Ä Å ô ö ò û ù Ö Ü ø á í ó ú ñ Ñ Ì ß
0 1 2 3 4 5 6 7 8 9 ¯ _
```

Names of workspaces, functions, variables, operators, and labels may be formed of any sequence of the above characters, except that the characters on the last line above cannot *start* a name.

Valid names include:

```
A
ABc
SALES_REPORT
TAX1984
∆
```

Here are some examples of **Invalid names**:

```
A B
```
contains a space
```
1984TAX
```
starts with a numeric digit
```
_REPORT
```
starts with "_"
```
DATA.3
```
"." isn't allowed

The environment in which APL2 operations take place is bounded by a *workspace*. The same name may be used to designate different objects (that is, variables, functions, operators, and labels) in different workspaces, without interference. It is also possible for a workspace to have the same name as an object it holds.

The workspace name is normally limited to eight characters, and the characters must form a valid APL name. (By using a special syntax, though, it is possible to use any name supported by the operating system as a workspace file name.)

The names of variables, functions, operators, and labels, however, may be of *any* desired length - up to 255 characters, which is probably longer than any name you would ever want to deal with.

Numbers

All numbers entered or displayed are in decimal, either in conventional form (including a decimal point if appropriate) or in *scaled form*. The scaled form consists of an integer or decimal fraction called the *multiplier* followed immediately by an "E" and then by an integer (which must not include a decimal point) called the *scale*. The scale specifies the power of ten by which the multiplier is to be multiplied. Thus `1.44E2` is equivalent to `144`.

APL2 supports complex numbers, which are normally represented with a "J" separating the real and imaginary parts. Polar forms are also available, with the angle expressed in either radians or degrees. For example, the square root of negative one may be entered as `0J1` in its standard form, as `0R1.570796327` in polar radian form, or as `1D90` in polar degree form. Complex numbers are always displayed using the `J` form.

Negative numbers are represented by an overbar immediately preceding the number. For example, `¯1.44` and `¯144E¯2` are equivalent negative numbers. Note that the overbar (`¯`) used to start a negative number differs from the bar (`-`) that denotes the subtract and negative functions. This avoids ambiguity in use of optional spaces when entering data.

```
      5 3
5 3
      5 ¯3
5 ¯3
      5-3
2
      5 -3
2
```

Conversely, the overbar cannot be used as a function:

```
      X←5
      Y←3
      X-Y
2
      -X
¯5
      X¯Y
VALUE ERROR
      X¯Y
      ^
      ¯X
SYNTAX ERROR
      ¯X
      ^
```

`X¯Y` is a valid name, but does not currently have a value. `¯Y` is not a valid name, but is treated as a single token, because its characters are all alphanumeric.

Functions

The word *function* derives from a word that means to execute or to perform. A function executes some action on its *argument* (or arguments) to produce a *result* that may serve as an argument to another function. For example:

```
            3×4
      12
            2+(3×4)
      14
            (-6)÷3
      ¯2
```

Functions represented by symbols, such as `+`, `-`, `×` and `÷`, are called *primitive functions*. They are automatically available for use in any workspace without having to copy them from somewhere.

Other functions are represented by names. They include user-defined functions (which are APL programs), and *system functions*. System functions, like primitive functions, are automatically available for use in any workspace, but they have *distinguished names* that begin with the `⎕` character, followed by a sequence of letters. For example, `⎕DL` is a system function used to wait (delay execution) for a specified length of time.

A function that takes one argument, such as the `-` in `(-6)` above, is said to be *monadic*; a function that takes two arguments, such as the *times* function, is said to be *dyadic*. Monadic functions always have their argument on the right in APL. Dyadic functions (whether primitive, system, or user-defined) take a right argument and a left argument.

In all cases, the same symbol or name can represent both monadic and dyadic functions. For example, `X-Y` denotes *subtraction* of `Y` from `X` (a dyadic function), and `-Y` denotes *negation* of `Y` (a monadic function). In fact APL2 assumes that every function defined to be dyadic can also be called monadically. This characteristic is called *ambi-valence*.

User functions can also be defined to be *niladic* (taking no argument). There are no primitive or system functions which are niladic, and the system treats a niladic function much as if it were a variable.

See also Terminology: Functions versus Operators.

Operators

The normal operation of a function may be altered by applying an *operator* to it. For example, + and × are primitive functions which apply independently to each element of an array.

```
      A←5 6 7
      B←2 3 4
      A+B
7 9 11
      A×B
10 18 28
```

Applying the / operator to produce +/ and ×/ modifies their normal operation, creating *derived functions* which apply between the items of a single argument. Using the same data as in the previous example:

```
      +/A
18
      ×/B
24
```

The / operator can be used with any dyadic function. Operators apply equally to user-defined functions, and, in fact, the operators themselves may be user-defined.

See also Terminology: Functions versus Operators.

Terminology: Functions versus Operators

Computer languages have sometimes created confusion between the terms *function* and *operator*. In APL, it's important to differentiate the terms.

- A *function* takes *data* objects as *arguments* and returns new *data* as a result.
- An *operator* takes *functions* (or occasionally data objects) as *operands* and returns a new function, called a *derived* function. The new derived function acts like any other function.

Note that we referred to operators as taking operands, and functions as taking arguments. This distinction is mostly useful when identifying the way that a token is being used. An operator always has an operand on its left. If it is a dyadic operator, it will have a second operand on its right. You can, if you wish, put parentheses around the operator and its operands. Once you have done that, you will see the arguments to the derived function outside the parentheses.

For example, ⍴ is a function called Reshape which accepts a left argument telling it how it should reshape its right argument. ¨ is an operator called Each which applies its operand to each item of the derived function argument(s).

```
      3⍴4     ⍝ Create 3 instances of the number 4
4 4 4
      2 3(⍴¨)4 5
 4 4  5 5 5
      2 3⍴¨4 5
 4 4  5 5 5
```
The last two expressions are exactly equivalent. The parentheses just help us see that ⍴ is an operand, while 2 3 and 4 5 are arguments. APL2 itself doesn't need the parentheses to recognize that.

Data

Data used in APL2 is one of two types, either *numeric* or *character*. Data is produced by:

- Explicit entry at the keyboard.
- Execution of APL2 functions and operators.
- Use of shared variables, system variables, system functions, and system commands.

The following topics provide information about data:

- Arrays
- Rank and Shape
- Variables and Constants
- Bracket Indexing
- Index Origin
- Adding More Structure to Arrays

Arrays

APL2 functions apply to collections of individual data items called *arrays*. An array is an ordered collection of items arranged along rectangular dimensions (called *axes*). The items of the array are numbers, characters, or other arrays, in any combination. For example, an array might be a list of three items, where the first is a character, the second a number, and the third a character string (which is really a subarray containing a list of characters).

Rank and Shape

The *rank* of an APL2 array is the number of dimensions or axes that it has. If you are familiar with some other computer languages, you may be thinking of the term *dimension* as the amount of data that can be stored; that's *not* what we mean here. We are referring to the axes, not the *length* of the data. For our purposes, a *dimension* and an *axis* are synonymous. When we want to refer to the amount of data *along* each of those dimensions, that's what we will call *shape*.

For example, a simple list of numbers has only one dimension - only length - and therefore is of rank one:

```
      V←2 3 5 7 11 13 17 19
      V
2 3 5 7 11 13 17 19
```

In APL, data in a list form like this is referred to as a *vector*.

An example of a rank-two object would be a table of numbers:

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

(We'll talk about ⍴ and ⍳ in a moment)

In APL2, two-dimensional data like this is referred to as a *matrix*.

Either of these examples could just as easily have used character data, or a mixture of numeric and character data.

A *scalar* has no dimensions and is of *rank zero*. APL2 supports arrays of up to rank 64, though most of us have trouble visualizing anything beyond rank 3 or 4.

The *shape* of an array may be measured by using the *shape* function, denoted by the **rho** (ρ) symbol:

```
      V
2 3 5 7 11 13 17 19
      ρV
8
      A←'ABCDEF'
      ρA
6
```

The shape function returns a count of the number of items along each of the dimensions. In the case of the vectors above, there was only one dimension. A matrix, because it is two-dimensional, will return two numbers:

```
      N
1  2  3  4
5  6  7  8
9 10 11 12
      ρN
3 4
      M
HELLO
THERE
      ρM
2 5
```

We showed an example above of how a vector is entered at the keyboard, but a matrix cannot be directly entered. You'll have to use a function to tell APL2 the shape that you want. A matrix is commonly formed by listing the items of data that the matrix is to contain, and then using the *reshape* function to create the desired shape. The reshape function uses the same rho symbol that the shape function uses, but has a left argument stating the desired resultant shape. (Note that the two functions are closely related. The *result* of monadic ρ for an array shows the *left argument* of the dyadic ρ required to create that array.)

The numeric matrix shown above, for instance, could be formed like this:

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

The number of numbers used to the left of the ρ-symbol determines the rank of the object being formed. Here, the two numbers `3 4` create a rank two object - a matrix. In a similar fashion, the rank of an object may be measured by counting the number of numbers that are returned with the monadic use of the ρ-symbol... in other words, measuring the shape of the shape:

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

```
9 10 11 12
      ⍴N
3 4
      ⍴⍴N
2
```

The right argument for the reshape function may be in any form. It could be a directly-entered list of items as we discussed above, or it could be data already stored under a name:

```
      V←2 3 5 7 11 13 17 19
      M←2 4⍴V
      M
 2  3  5  7
11 13 17 19
      A←3 2⍴'ABCDEF'
      A
AB
CD
EF
      B←2 4⍴A
      B
ABCD
EFAB
```

The rank and shape of the right argument to ⍴ are of no concern. Its data items are just used in sequence to build the result array. If the right argument has extra items, they are ignored. If it doesn't have enough (as in the last example above) the function simply goes back to the beginning of the array, and continues selecting items as it needs.

Arrays of arbitrary shape and rank may be produced by the same scheme. For example:

```
      T←2 3 4⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
      T
ABCD
EFGH
IJKL
MNOP
QRST
UVWX
      ⍴T
2 3 4
```

This three-dimensional array has two planes, each with three rows and four columns. Three-dimensional arrays are displayed with a blank line separating the planes, and higher-dimensional arrays simply extend this scheme.

Variables and Constants

An array that is stored under a name is called a *variable*, because its value may be varied at any time simply by reassigning a new value to the name. All of the names that we have shown in the discussion of Rank and Shape are variables.

A *constant* is a number or string of numbers or a character or string of characters that appears explicitly in an APL2 expression.

A single number entered by itself is accepted by the system as a *scalar*. A constant *vector* may be entered by listing the numeric components in order, separated by one or more spaces.

A scalar character constant may be entered by placing the character between single quotation marks (as in `'A'`), and a character vector may be entered by listing the characters between single quotation marks (as in `'THIS IS TEXT'`). The blanks are part of the data, and are treated like the other characters, but the enclosing quotation marks are *not* part of the data. That last example is twelve characters long, because it includes the two blanks, but not the two quotation marks.

To include a single quotation mark character within a character constant you must enter it as a pair of single quotation marks. Thus, the contraction of `CANNOT` is entered as `'CAN''T'`. APL2 displays it as `CAN'T`, and it consists of five characters.

There are no special considerations for entering double quotation mark characters:

```
      'Do you know what "rank" means?'
Do you know what "rank" means?
      'No, I don''t'
No, I don't
```

Bracket Indexing

Note: The examples here assume the default index origin.

The items of an array may be selected by *bracket indexing*. For example:

```
      V←2 3 5 7 11 13 17 19
      V[3 1 5]
5 2 11
      (2 3 5 7 11 13 17 19)[3 1 5]
5 2 11
      A←'ABCDEFGH'
      A[8 5 1 4]
HEAD
      'ABCDEFGH'[8 5 1 4]
HEAD
```

The numbers within the square brackets indicate the positions of the data that is being selected. If *any* of the indices are out of range, you'll get an error message:

```
      'ABCDEFGH'[8 5 1 35]
INDEX ERROR
      'ABCDEFGH'[8 5 1 35]
      ^
```

Elements may be selected from any array by indexing in the manner shown for vectors, except that indices must be provided for each dimension.

```
      M
 2  3  5  7
11 13 17 19
      M[2;3]
17
      M[2 1;2 3 4]
13 17 19
 3  5  7
```

Note: Scalars are a special case, because they have no dimensions. You can't use bracket indexing for them, but you can use the similar "Index" (⌷) function.

That last example introduced the idea of using bracket indexing to select a *cross section* of the array. All rows (listed before the semicolon) and all columns (listed after the semicolon) are selected, and then arranged in the order specified. Of course you can select cross sections of arrays of any dimension. The shape of the result will directly correspond to the shapes of the vectors used to select the items.

```
      ⍴M[2 1;2 3 4]
2 3
      T
ABCD
EFGH
IJKL
MNOP
QRST
UVWX
      T[2;1;4]
P
      T[2;1 2 3;1 2 3 4]
MNOP
QRST
UVWX
      ⍴T[2;1 2 3;1 2 3 4]
3 4
```

As a special convention, you can omit the indices along one or more axes, but not the associated semicolons, to select all items in order along that axis. Thus the last examples above can be written more simply as

```
      T[2;;]
MNOP
QRST
UVWX
      ⍴T[2;;]
3 4
```

Index Origin

The indexing used in examples throughout this document is called *origin `1`* because the first element along each axis (or dimension) is selected by the index `1`. This is the default for a new workspace, but you may also use *origin `0`* indexing by setting the *index origin* to `0`. The index origin is controlled by a *system variable* denoted by `⎕IO`. Thus:

```
      V←2 3 5 7 11 13 17 19
      ⎕IO←1
      V[1 2 3]
2 3 5
      ⍳3
1 2 3
      ⎕IO←0
      V[0 1 2]
2 3 5
      ⍳3
0 1 2
```

The `⍳` function shown above is called *Interval*, because it produces a progression of integers. The first integer in the progression is `⎕IO`, and the interval between values is `1`.

In APL2, you always have the choice of using either origin 1 or origin 0. You may find that the use of origin 0 may make some applications easier to write. This is especially true where certain mathematical operations are being performed. Calculations involving number-base conversions, for example, are often cleaner if you're working in origin 0. Some indexing operations themselves are also a little cleaner. For example:

```
      N
1  2  3  4
5  6  7  8
9 10 11 12
      ⎕IO←1
      '∘⎕'[1+N>6]
∘∘∘∘
∘∘⎕⎕
⎕⎕⎕⎕
      ⎕IO←0
      '∘⎕'[N>6]
∘∘∘∘
∘∘⎕⎕
⎕⎕⎕⎕
```

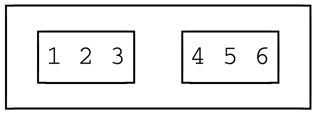
However, origin 0 can also be confusing at times, simply because most of us grew up being accustomed to thinking of series of numbers as starting with one instead of zero. [Neither of these is correct, of course; in our hearts we all know that the number series really begins at negative infinity.] Years of seeing lists numbered "1, 2, 3" instead of "0, 1, 2" tends to leave its mark. Throughout our lives we have seen that

1. House numbers start with 1 (Number 0 Downing Street?)
2. Magazine page 0 is consistently missing
3. Days of the month start with 1 (it's *really* hard to find an exception here)

So, rather than complicating your life by bucking this ingrained bias, APL2 uses origin 1 as its default. You can always change it, but that's what you'll see in any new workspace.

Adding More Structure to Arrays

Let's assume that we have an array named A, which contains two pieces of data: a string of numeric data having the value 1 2 3, and a similar string of numeric data having the value 4 5 6. A then can be represented as a *two-item* vector. We can think of the array as looking like this:



The outer box represents A, which contains two items. Each item is a three-item vector. In APL2, an item of an array can be any other array. Arrays containing items which are other arrays are called *nested arrays*.

It's very easy to create such an array in APL2. Just use parentheses to group the items:

```
      MORE←(1 2 3) (4 5 6)
      ⍴MORE
2
      MORE
 1 2 3  4 5 6
```

ρMORE tells us that the shape of MORE is 2; i.e. that it has two items. Note that when APL2 displays a nested array it adds a blank in front of the first item in each nested group. You can also request a display with boxes by using a DISPLAY function which we provide. (See DISPLAY under workspaces in the *APL2 User's Guide*.)

```
      )COPY 1 DISPLAY DISPLAY
      DISPLAY MORE
┌→────────────────┐
│ ┌→────┐ ┌→────┐ │
│ │1 2 3│ │4 5 6│ │
│ └~────┘ └~────┘ │
└∊────────────────┘
```

The degree of nesting of an array is called *depth*. A simple scalar has a depth of 0. The simple vector (or even an n-dimensional array) has a depth of 1. This means that all of its items are simple scalars, that is, either single numbers or single characters. The depth of MORE above is 2. A depth of 2 means that at least one of its items has a depth of 1.

APL2 provides a depth function (≡) which shows the depth of an array.

```
      ≡1 2 3
1
      ≡MORE
2
```

The matrix M, below, shows the use of nested arrays to add headings to a table and to substitute 'NONE' for items whose value is 0. The matrix has five rows and three columns. Each item in the first row is a character vector, and each item in the first column is a character vector. NONE in the last row, last column, is also a character vector. The depth of M is 2.

```
      M
FOOD   CALORIES  PROTEIN
milk        160        9
apple        60        1
bread        75        2
jelly        50     NONE
```

Assigning Values to Names

The left arrow is used to assign a value to a name. As we have usually shown it throughout this material, the arrow is near the left end of an APL statement, with only the associated name to its left.

```
      HYPOTENEUSE←((LEG1⋆2)+(LEG2⋆2))⋆.5
```

When used this way the APL statement evaluates the expression and assigns the result to the name *instead* of displaying the result.

But a specification arrow (as the left arrow is often called) can be used in the middle of an APL statement as well. In this case the value assigned to the name is made available for additional operations. Consider a function called REPORT_SALES which takes a region number as an argument; and another function called REGION_FOR which takes a city name and returns a region number. You might use these functions as follows:

```
      REPORT_SALES REGION_FOR 'Chicago'
```

But if you needed the region number for several statements, you might instead decide to do something like:

```
      REGION←REGION_FOR 'Chicago'
      REPORT_SALES REGION
```

Many APL users prefer to keep the statements simple, with the names assigned at the left end of the statements. But it is also legal to combine the above two statements into one:

```
      REPORT_SALES REGION←REGION_FOR 'Chicago'
```

In this statement `REPORT_SALES` is the last operation performed. Since the last operation is not specification, a result will be displayed if `REPORT_SALES` produces one.

Any number of assignment arrows may occur in an expression. Displaying any *intermediate result* in an APL2 expression can be obtained by including the characters "⎕←" to the left of any portion of the expression. For example:

```
      A←2+⎕←3×B←4
12
      A
14
      B
4
```

The cases covered so far are called *simple specification* (or *simple assignment*). The reason for that becomes obvious as you look at some of the other things you can do with a specification arrow:

- Indexed Specification
- Vector Specification
- Selective Specification

Indexed Specification

If the token to the left of a specification arrow is a right bracket, the bracket operation is first performed, then the value to the right of the specification arrow is assigned to the items selected by the bracket indexing.

```
      A←1 2 3
      A[2]←4
      A
1 4 3
      A[2 3]←5 6
      A
1 5 6
      A[2 3]←2
      A
1 2 2
```

Note that the bracket operation can produce either a single item or multiple items. When multiple items are produced, the shape must be *conformable* with the expression on the right, which means that either the rank and dimensions must agree, or that the right expression must be a single item so that it can be extended to all selected items, as in the last example above.

Vector Specification

You can use a single left arrow to assign values to a set of names from the items of a vector:

```
      (A B)←14 4
      A
14
      B
4
```

The shape of the vector has to agree with the number of names being assigned; and normally the expression has to be a vector. But there is one case where APL even relaxes that rule:

```
      (MY THREE VARIABLES)←0
      MY
0
      THREE
0
```
(Talk about a confusing choice of names.)

The trick there is that the expression to the right of the left arrow is a scalar rather than a vector, so it has no shape. This is an example of an APL principle called *scalar extension*. Having no shape, the scalar can take on whatever shape it is needed for.

The expression on the right also need not be a *simple* vector. It can be an arbitrarily nested array:

```
      (ADR¯NUM ADR¯STREET)←555 ('Bailey' 'Avenue')
      ADR¯STREET
 Bailey Avenue
      ⍴ADR¯STREET
2
```

Selective Specification

If the token to the left of a specification arrow is a right parenthesis, the expression within the paired parentheses is performed symbolically (but not actually evaluated), and the value to the right of the arrow is assigned to the resulting item or items. This is called *selective specification* and has a number of restrictions. The supported functions are:

Monadic	`↑   ⍉   ,   ⌽   ,[X]   ⌽[X]`
Dyadic	`[L]   ⊃   ⍴   ⍉   ↑   ↓   ⌷   ⌽   ↑[X]   ↓[X]   ⌷[X]   ⌽[X]`
Derived	`LO\   LO/   LO\[X]   LO/[X]`

Selective specification is normally used to replace selected items within an array. In this case the shape of the array containing replacement data must agree with the shape of the selected items.

Sometimes selected items are not simple data items, but are themselves subarrays. When a whole subarray is replaced as an item, the structure of the replacement item is not relevant. The replacement could have a different shape, rank, or depth from the item it is replacing.

In ordinary cases, selective specification can be understood if you understand how the selection expression works when it is not on the left of an assignment. For example:

```
      V← 10 20 30 40
      2↑V
10 20
      (2↑V)←100 200
      V
100 200 30 40
```

The function *Take* does not select the first two items of `V`, it selects the locations of the first two items of `V`. This resulting vector of locations is considered a simple vector even if the items at those locations are deeply nested. The data on the right of the assignment then replaces data at those locations.

Various selections and replacements are shown below for the matrix `M`. The examples assume that each selective specification expression uses the *original* specification of `M`.

```
      M←3 4ρ'ABCDEFGHIJKL'
      M
ABCD
EFGH
IJKL
      (2 3⌷M)←'⎕'
      M
ABCD
EF⎕H
IJKL
      (1(2 4)⌷M)←'∇⋆'
      M
A∇C⋆
EF⎕H
IJKL
      (2 1↑M)←'⊖⌹'
      M
⊖∇C⋆
⌹F⎕H
IJKL
      (,M)←⍳12
      M
1  2  3  4
5  6  7  8
9 10 11 12
      M←3 4ρ'ABCDEFGHIJKL'
      (4↑,⍉M)←'○⋆÷⎕'
      M
○⎕CD
⋆FGH
÷JKL
```

The last example above demonstrates the application of several functions in selective specification. The positions replaced were the first four taken in row-major order after `M` was transposed (its rows and columns interchanged). These are the characters `AEIB`, which are then replaced with the `○⋆÷⎕`, respectively.

The example below shows that scalars being selectively assigned to a non-scalar array of locations are replicated as necessary.

```
      M←3 4⍴'ABCDEFGHIJKL'
      ((1 3)(1 4)⌷M)←'*'
      M
*BC*
EFGH
*JK*
```

Restrictions

If `B` is a shared variable, then

```
      ((B=' ')/B)←'*'
```

is an error because the leftmost mention of `B` is a reference of the shared variable and causes `B` to receive a new value.

Order of Evaluation

In APL2, the order of evaluation is from right to left, with a few qualifications that we will get to in a moment. In particular, there is *no* hierarchy among the functions, such as multiplication being executed before addition. All functions are treated alike. This should come as a real relief if you have used other programming languages. (The C language, for example, has 15 levels of precedence!)

It would be confusing enough to try to define a hierarchy among the very complete set of primitive functions provided by APL2, but even that wouldn't begin to be enough. APL2 allows you to define your own functions, which behave just like the primitive functions. Would you appreciate having to specify a precedence for every routine you write, compared with every other routine you or anyone else has ever written?

So, despite the initial shock at learning that `2×3+4` is `14` in APL (right to left, not multiplication first), it is soon reassuring to know without worrying about it that in

```
      2 FOO 3 + 4
```
the addition happens before calling `FOO`, while in

```
      2 + 3 FOO 4
```
the addition happens last.

Exceptions to the Rule

Now we take up the few qualifications which modify the basic right-to-left order. In order of execution priority they are:

1. Parentheses
2. Bracket notation.
3. Specification object.
4. Vector notation.

5. Operand binding.

followed by the basic rule,

6. Normal function processing order.

Parentheses

Parentheses are used in the familiar way to control the order of evaluation in a statement. Any expression within matching parentheses is evaluated before applying any function to the result outside the matching pair. Parentheses are always permissible if they are properly paired and what is inside evaluates to an array, a function, or an operator. There are no restrictions on levels of nesting.

Parentheses are often used to surround the left argument of a function, so that it is evaluated in one complete piece. For example, `(7-3)×2` is `8` whereas `7-3×2` is `1`.

Parentheses are not needed for the right argument of a function, because of APL's normal evaluation order. This, combined with the careful design of many APL functions so that the primary data is in the right argument, means that fewer parentheses are required in APL than in most languages.

Priority of Bracket Operations

Brackets provide a very useful notation, but one that does not fit in perfectly with the rest of APL syntax. Like parentheses, brackets completely override the order of processing for the expressions contained within them. But unlike parentheses, brackets are an implied function or operator (depending on what is to the left of the left bracket), and are tightly bound to the closest function or data that can be identified on their left.

For discussion of brackets as a function, see Bracket Index. Use of brackets as an operator is restricted to specific primitive and derived functions, and is discussed with those functions or the operators deriving them, as a "with axis" variant.

The tight binding of bracket notation to the token on the left takes priority over all other language constructs except parentheses.

```
      A←1 2 3
      A[2]←4
      A
1 4 3
      A A[2]
 1 4 3  4
      1 1 2 3 5 8[4]
RANK ERROR
      1 1 2 3 5 8[4]
      ^          ^
      (1 1 2 3 5 8)[4]
3
```

Priority of Specification

Normally the token to the left of a left arrow is the name of a variable, or an unused name that can become the name of a variable. The left arrow is called *Specification* or *Assignment*, and associates the name on its left with the evaluated expression on its right. Although Specification is not, formally, a function or operator, it does make the value of the expression to its right available for further use in the APL expression, much as if it were a function result. (But this is done *only* if the value is required. Unlike a real function, if assignment is the last operation the value is not displayed.)

Two constructions can be used to modify this normal behavior:

- If the token to the left of a specification arrow is a right bracket, indexed specification is performed.
- If the token to the left of a specification arrow is a right parenthesis, selective specification is performed.

In either case, note that if the value is used further in the expression, it is the value to the right of the specification arrow that is propagated, not the value of the named variable or variables.

```
      A←1 2 3
      2+A
3 4 5
      2+A[2]←10
12
      B←'HELLO'
      2+(A B)←1 2
3 4
      A
1
      B
2
```

Vector Notation and its Priority

A series of value expressions separated from each other by nothing but spaces is treated as a vector. Here are some examples:

```
      2 3 4
      'A' 'B' 'C'
      1 (2 3) 4
```

In that last example, `(2 3)` is a value expression which is then combined with the remainder of the expression to form a three-item nested array. Here are some additional examples which extend that concept:

```
      2 'ABC'
      1 (2+3) 4
      AA←2 3
      1 AA 4
```

Note that the last example has exactly the same value as the last example in the previous group.

Order of Evaluation

Like other APL expressions, the items within a vector notation are evaluated from right to left:

```
      A←1
      A (A←2) A
2 2 1
```

But forming a vector using vector notation takes precedence over applying an array to the function or operator on its right.

```
      1 2 3 + 4 5 6
5 7 9
      A←1
      3 A + 1 1
4 2
```

Vector notation does *not* take precedence over assignment.

```
      A←1
      A B←2 3
 1  2 3
      A
1
      B
2 3
      (A B)←2 3
      A
2
      B
3
```

Nor does it take precedence over bracket indexing.

```
      A←1 2 3
      A A[2]
 1 2 3  2
```

Operand Binding

Monadic Operators

Most operators are monadic; which means they take an operand on the left. There is little chance of confusion with these. Working from right to left, APL2 finds an operator, then looks to its left for an operand.

- If it finds an array, it first applies the vector notation rule, then uses the resulting array as an operand.
- If it finds a function, that becomes the operand.
- If it finds another operator, APL2 knows that it will first have to produce a derived function from that operator, and the derived function will then become the operand for the first operator.

An example will make the last case clear. The Each (¨) operator applies its operand function *to* each item (at the top level) of the data argument. The Reduce (/) operator applies its operand function *between* each item (at the top level) of the data argument. Thus:

```
      ×/2 3 ¯1
¯6
      ×¨2 3 ¯1
1 1 ¯1
```

In the first case × is applied between each item, so is used as a multiplication:

```
      2×3×¯1
¯6
```

In the second case × is applied to each item, so it becomes a monadic function which returns the sign of the value:

```
      (×2) (×3) (×¯1)
1 1 ¯1
```

(Actually, the function would do that even without the ¨. See Scalar Functions.)

All that was background. Now we are ready for the real example:

```
      ×/¨(2 3 ¯1) (2 3 4)
¯6 24
```

Looking from right to left we can see that APL2 will first create two vectors and then combine them into one nested vector. Now it encounters the ¨ operator, so it looks to its left. It sees /, which is also an operator; so cannot be an operand. Continuing left it sees that it can form the derived function ×/, which is a product reduction. It can then use that function as the operand for ¨, like this:

```
      (×/2 3 ¯1) (×/2 3 4)
¯6 24
```

Dyadic Operators

APL2 includes only one primitive dyadic operator, the Array Product operator, but users can define additional dyadic operators. The Array Product operator, when used with a left and right function, produces any of an infinite variety of Inner Product derived functions. The specific inner product derived function corresponding to the original mathematical use of the term is +.×, which forms products of pairs of values from the left and right argument data, and sums groups of those products.

```
      A←2 3⍴⍳6
      B←3 4⍴⍳12
      A
1 2 3
4 5 6
      B
1  2  3  4
5  6  7  8
9 10 11 12
```

Multiplying the first row of A by each of the first two columns of B gives:

```
      +/1 2 3 × 1 5 9
38
      +/1 2 3 × 2 6 10
44
```

+.× does that for each row of A combined with each column of B:

```
      A+.×B
38 44  50  56
83 98 113 128
```

The point to focus on right now is not why anyone would want such an oddly defined facility (though it does turn out to be extraordinarily useful), but why APL2 analyzes the syntax of the expression the way it does. Note that the right-most part of the expression is `×B`, which APL2 is quite capable of evaluating on its own:

```
      ×B
1 1 1 1
1 1 1 1
1 1 1 1
```

Not a terribly exciting result, but it does say that all items of `B` are positive.

But APL2 does *not* evaluate the expression that way. This is the first real example that we have seen showing that operands take priority over normal right-to-left function processing. When APL2, scanning right to left, encounters the ×, it has to look further anyway to find out whether there is a left argument, i.e. to decide whether × is the monadic Direction function or the dyadic Product function. When it finds the dyadic operator symbol `.` it knows that × is being used as an operand.

To carry this one step further, `/` which we also used above, is a monadic operator. How should APL2 evaluate an expression like `+.×/`? `×/` is a derived function which could become the right operand to `+.`, but `+.×` is also a derived function which could become the left operand to `/`. The answer is that the right operand of an dyadic operator is bound more tightly than the left operand of an operator. i.e.,

```
      +.×/      ⍝ Is equivalent to
      (+.×)/    ⍝ this
      +.(×/)    ⍝ not to this
```

Errors

Entry of a statement that cannot be executed will invoke an *error report*. Newcomers to APL2 often tend to worry needlessly about typing inputs that result in errors. These error reports are some of the most *helpful* aids that you could ask for toward learning the language. APL2 error reports are designed to be clear, concise, and precise.

As opposed to doing everything that you could do to prevent generating errors, it may be helpful to deliberately try out many of the error conditions. This is a good way of learning how various functions are defined. Learning by doing is always preferable. And don't worry that you may enter something that you shouldn't have... nothing that you can enter can hurt the machine. This gives you full freedom to experiment.

An APL2 error report indicates the nature of the error and displays carets, indicating both where the error occurred and where the execution halted. For example:

```
      B←1 2 3 + A←4 5
LENGTH ERROR
      B←1 2 3+A←4 5
        ^     ^
```

There's a wealth of information available from these error reports. Let's see just what this message is telling us:

```
      B←1 2 3 + A←4 5    Here is the line that you typed in.

LENGTH ERROR             Two lengths are incompatible

      B←1 2 3+A←4 5      Here is your input line
                         (unneeded blanks have been removed)

       ^     ^           Scan position and error position
```

There will typically be two carets under the line of code. The *left caret* shows you how far APL2 got in its right-to-left scan of the line (here, the system has recognized that `1 2 3` is a numeric vector to be treated as a left argument, and it has stopped at the assignment arrow, which it has *not* yet processed). The *right caret* shows you the point of the actual error. Normally, that will indicate which function APL2 was evaluating when the error occurred. In this example, the arguments to the + function aren't compatible with each other, so the requested addition can't be performed.

Sometimes one caret will be "on top of" the other, so it will look like there is only one.

Primitive Functions

In this section we use `L` to refer to the left argument, `R` to refer to the right argument, and `X` to refer to an axis value.

You will find below a complete Alphabetic List of primitive function names, from each entry of which you can get to information about that function. In addition, you can use any of the following lists to get to that same information:

Primitive Function Symbols
: This list shows which symbols are used monadically and dyadically, and the function names associated with each usage.

Scalar Functions
: Functions which transform simple scalar values within any array, no matter how the data is structured, and produce results whose structure matches their arguments.

Structural Functions
: Functions which retain part or all of the data in their arguments, but produce results whose structure has been changed.

Information Functions
: Functions which return information about the structure or content of their arguments, but do not return the data itself.

Non-scalar transform Functions
: Functions which transform data and either work on certain types of structures, or return data structures different from their arguments.

Alphabetic List of Functions

- Add
- And
- Binomial
- Bracket Index
- Catenate
- Catenate with Axis
- Ceiling
- Circle Functions (trigonometric)
- Conjugate
- Deal
- Decode
- Depth
- Direction (sign)
- Disclose
- Disclose with Axis
- Divide
- Drop
- Drop with Axis
- Enclose
- Enclose with Axis
- Encode
- Enlist (structure to vector)
- Equal
- Execute
- Exponential
- Factorial
- Find
- First
- Floor
- Format (default)
- Format by Example
- Format by Specification
- Grade Down
- Grade Down with Collating Sequence
- Grade Up
- Grade Up with Collating Sequence
- Greater Than
- Greater Than or Equal to
- Index (select items)
- Index with Axis
- Index of
- Interval
- Laminate (add dimension)
- Less Than
- Less Than or Equal to

- Logarithm
- Magnitude (absolute value)
- Match
- Matrix Divide
- Matrix Inverse
- Maximum
- Member
- Minimum
- Multiply
- Nand
- Natural Logarithm
- Negative
- Nor
- Not
- Not Equal
- Or
- Partition
- Partition with Axis
- Pi Times
- Pick
- Power
- Ravel (array to vector)
- Ravel with Axis
- Reciprocal
- Reshape
- Residue (remainder)
- Reverse
- Reverse with Axis
- Roll
- Rotate
- Rotate with Axis
- Shape
- Subtract
- Take
- Take with Axis
- Transpose
- Transpose (reversed axes)
- Without

Names and Valences of Primitive Function Symbols

Symbol	Monadic Use	Dyadic Use
+	Conjugate	Add
-	Negative	Subtract
×	Direction	Multiply
÷	Reciprocal	Divide
⋆	Exponential	Power
|	Magnitude	Residue
⌈	Ceiling	Maximum
⌊	Floor	Minimum
⍟	Natural Logarithm	Logarithm
?	Roll	Deal
!	Factorial	Binomial
○	Pi Times	Circle Functions
⌹	Matrix Inverse	Matrix Divide
∧		And
∨		Or
⍲		Nand
⍱		Nor
∼	Not	Without
<		Less Than
≤		Less Than or Equal
=		Equal
≥		Greater Than or Equal
>		Greater Than
≠		Not Equal
≡	Depth	Match
,	Ravel	Catenate or Laminate
⍴	Shape	Reshape
⊃	Disclose	Pick
⊂	Enclose	Partition
⌽	Reverse	Rotate
⍉	Transpose (reversed axes)	Transpose
↓		Drop
↑	First	Take
[ ]		Bracket Index
⌷		Index
⍋	Grade Up	Grade Up with Collating Sequence
⍒	Grade Down	Grade Down with Collating Sequence

Symbol	Monadic Use	Dyadic Use
⍳	Interval	Index of
⍷		Find
∊	Enlist	Member
⍎	Execute	
⍕	Format (default)	Format by Example or Format by Specification
⊥		Decode
⊤		Encode

Scalar Functions

Monadic

These functions apply to scalar items within an arbitrarily structured array, and produce a result with the same structure.

Note: Trigonometric functions are provided as dyadic Circle Functions.

Syntax	Name
`-R`	Negative
`÷R`	Reciprocal
`⌈R`	Ceiling
`⌊R`	Floor
`∣R`	Magnitude (absolute value)
`×R`	Direction (sign)
`+R`	Conjugate
`⋆R`	Exponential
`⍟R`	Natural Logarithm
`!R`	Factorial
`○R`	Pi Times
`?R`	Roll
`~R`	Not

Dyadic

These functions apply to scalar items within any pair of arbitrarily but equivalently structured arrays, and produce a result with the same structure. They also permit scalar extension, replicating a scalar item to agree with the shape of the other argument.

Syntax	Name
`L+R`	Add
`L-R`	Subtract
`L×R`	Multiply
`L÷R`	Divide
`L∣R`	Residue (remainder)
`L⌈R`	Maximum
`L⌊R`	Minimum
`L○R`	Circle Functions (trigonometric)
`L⋆R`	Power
`L⍟R`	Logarithm

Syntax	Name
`L∧R`	And
`L∨R`	Or
`L⍲R`	Nand
`L⍱R`	Nor
`L<R`	Less Than
`L≤R`	Less Than or Equal to
`L=R`	Equal
`L>R`	Greater Than
`L≥R`	Greater Than or Equal to
`L≠R`	Not Equal

Structural Functions

The following functions manipulate data structures without affecting the values of individual items.

Monadic

Syntax	Name
`↑R`	First
`⌽R`	Reverse
`⌽[X]R`	Reverse with Axis
`⍉R`	Transpose (reversed axes)
`,R`	Ravel (array to vector)
`,[X]R`	Ravel with Axis
`∊R`	Enlist (structure to vector)
`⊂R`	Enclose
`⊂[X]R`	Enclose with Axis
`⊃R`	Disclose
`⊃[X]R`	Disclose with Axis

Dyadic

Syntax	Name
`L⍴R`	Reshape
`L⌽R`	Rotate
`L⌽[X]R`	Rotate with Axis
`L⍉R`	Transpose
`L,R`	Catenate
`L,[X]R`	Catenate with Axis
`L,[X]R`	Laminate (add dimension)
`L~R`	Without
`L↑R`	Take
`L↑[X]R`	Take with Axis
`L↓R`	Drop
`L↓[X]R`	Drop with Axis
`L⌷R`	Index (select items)
`L⌷[X]R`	Index with Axis
`A[I]`	Bracket Index
`L⊃R`	Pick
`L⊂R`	Partition
`L⊂[X]R`	Partition with Axis

Information Functions

Monadic

Syntax	Name
`⍴R`	Shape
`≡R`	Depth
`⍋R`	Grade Up
`⍒R`	Grade Down

Dyadic

Syntax	Name
`L≡R`	Match
`L∊R`	Member
`L⍳R`	Index of
`L⍷R`	Find
`L⍋R`	Grade Up with Collating Sequence
`L⍒R`	Grade Down with Collating Sequence

Non-scalar transform Functions

Monadic

Syntax	Name
`⍳R`	Interval
`⍕R`	Format (default)
`⍎R`	Execute
`⌹R`	Matrix Inverse

Dyadic

Syntax	Name
`L⊥R`	Decode
`L⊤R`	Encode
`L?R`	Deal
`L⌹R`	Matrix Divide
`L!R`	Binomial
`L⍕R`	Format by Example
`L⍕R`	Format by Specification

Add

Sum of `L` and `R`.

```
      5+1 2 3
6 7 8
      1 2 3+4 5 6
5 7 9
```

`L`,`R`: Numeric

And

A `1` if both `L` and `R` are `1`; a `0` otherwise.

```
      0 0 1 1∧0 1 0 1
0 0 0 1
```

`L`,`R`: Boolean

Binomial

The number of combinations of `R` things taken `L` at a time.

```
      2!8
28
```

`L`,`R`: Numeric (If `R` is a negative integer, `L` must be integer.)

Bracket Index

The items of `A` specified by index arrays `I`.

```
      LANG←'APPLE PIE'
      LANG[1 7 4]
APL
      M1←2 2⍴⍳4
      M1[;2]
2 4
```

`A`: Nonscalar array
`I`: Simple array of nonnegative integers
Implicit argument: `⎕IO`

Catenate

`L` and `R` joined along the last axis.

```
      M1←2 2ρι4
      M2←2 2ρ'ABCD'
      M2,M1
AB 1 2
CD 3 4
```

Catenate with Axis

L and R joined along axis X.

```
      M1←2 2⍴⍳4
      M2←2 2⍴'ABCD'
      M2,[1]M1
A B
C D
1 2
3 4
```

X: Simple scalar, nonempty, nonnegative integer
Implicit argument: ⎕IO

See also Laminate for non-integral values of X.

Ceiling

For a real `R`: the smallest integer greater than or equal to `R`.

```
      ⌈.4 ¯.3 ¯3.4
1 0 ¯3
```

`R`: Numeric
Implicit argument: `⎕CT`

Circle Functions

L specifies the function to be performed on R. The values of L and the functions they represent are shown in the table below.

```
      1○1.5708
1
      2○.25
0.9689124217
```

R: Numeric
L: Integer; ¯12≤L≤12

L	Function	L	Function
		0	(1-R⋆2)⋆.5
¯1	Arcsin R	1	Sine R
¯2	Arccos R	2	Cosine R
¯3	Arctan R	3	Tangent R
¯4	(¯1+R⋆2)⋆.5	4	(1+R⋆2)⋆.5
¯5	Arcsinh R	5	Sinh R
¯6	Arccosh R	6	Cosh R
¯7	Arctanh R	7	Tanh R
¯8	-(8○R)	8	-(¯1-R⋆2)⋆.5 for R≥0 (¯1-R⋆2)⋆.5 for R<0
¯9	R	9	Real R
¯10	+R	10	|R
¯11	0J1×R	11	Imaginary R
¯12	⋆0J1×R	12	Phase R

Conjugate

R with the imaginary part negated.

```
      +.4 ¯5 3J4 ¯3J¯4
0.4 ¯5 3J¯4 ¯3J4
```

R: Numeric

Deal

L integers selected at random from ⍳R without replacement.

```
      ⎕RL←16807
      5?5
5 1 2 4 3
```

R: Simple scalar or 1-item vector; nonnegative integer less than 2⋆31
L: Simple scalar or 1-item vector; nonnegative integer less than or equal to R
Implicit arguments: ⎕IO, ⎕RL

Decode

The base 10 representation of `R`, where `R` is a number of base `L`. The value of a polynomial having coefficients `R` at point `L`.

```
      2⊥1 0 0 1
9
      60 60 60⊥1 30 20
5420
```

`L`,`R`: Simple numeric arrays

Depth

Levels of nesting of `R`: `0` for a simple scalar; `1` plus the depth of the deepest item for other arrays.

```
      ≡4
0
      ≡2 2⍴⍳4
1
      ≡2 2⍴1 2 3 (4 5)
2
```

Direction

For a real R: ¯1 if R is negative; 0 if R is zero; 1 if R is positive. For an imaginary R: the nonreal number of magnitude 1 that has the same phase as R.

```
      ×¯5 0 5
¯1 0 1
      ×3J4 ¯3J4
0.6J0.8 ¯0.6J0.8
```

R: Numeric

Disclose

`R` restructured into an array whose rightmost axes come from the axes of the items of `R`.

```
      ⊃(1 2 3)(4 5 6)
1 2 3
4 5 6
```

Disclose with Axis

R restructured into an array; X specifies the axes of the array into which items of R are structured.

```
      ⊃[1]('ABCD')('EFGH')
AE
BF
CG
DH
```

X: Simple scalar or vector; nonnegative integers
Implicit argument: ⎕IO

Divide

`L` divided by `R`.

```
      0 8÷0 0.4
1 20
```

`L`,`R`: Numeric

Drop

For a positive scalar `L`: removes the first `L` items of `R`.
For a negative scalar `L`: removes the last `L` items of `R`.

```
      2↓1 2 3 4
3 4
      ¯2↓1 2 3 4
1 2
```

`L`: Simple scalar or vector; integer

Drop with Axis

For a positive scalar L: removes the first L items of the Xth axis of R.
For a negative scalar L: removes the last L items of the Xth axis of R.

(In a matrix, for example, if X is 1, L rows are dropped from R; if X is 2, L columns are dropped from R.)

```
      H←2 4⍴'ABCDEFGH'
      2↓[2]H
CD
GH
```

L: Simple scalar or vector; integer
R: Nonscalar array
X: Simple scalar or vector; nonnegative integers
Implicit argument: ⎕IO

Enclose

A scalar array whose only item is `R`.

```
      R←2 3⍴⍳6
      ⊂R
1 2 3
4 5 6
      ⍴⊂R
      ≡⊂R
2
```

Enclose with Axis

An array whose items are those in the axes of R specified by X.

```
      R←2 3⍴'ABCDEF'
      ⊂[1]R
AD BE CF
      ⍴⊂[1]R
3
      ≡⊂[1]R
2
```

X: Simple scalar or vector; nonnegative integers
Implicit argument: ⎕IO

Encode

The base `L` representation of `R`, where `R` is a base 10 number.

```
      2 2 2 2 2⊤16
1 0 0 0 0
      60 60 60⊤5420
1 30 20
```

`L`,`R`: Simple numeric arrays

Enlist

A simple vector whose items are the simple scalars in `R`.

```
      R←9 (7 8) (2 3⍴⍳6)
      ∊R
9 7 8 1 2 3 4 5 6
```

Equal

A `1` if `L` is equal to `R`; a `0` otherwise.

```
       20 30 40=40 30 20
0 1 0
```

Implicit argument: `⎕CT`

Execute

The evaluation of the result of the APL2 expression represented by the character vector `R`.

```
      ⍎'3+4'
7
```

`R`: Simple character scalar or vector

If the argument is empty, or represents a defined function or operator without an explicit result, then `⍎R` has no result. Execute of a branch statement is permitted only if `⍎` is the leftmost operation on the line.

Exponential

e to the `R`th power.

```
      ⋆0 1 2
1 2.718281828 7.389056099
```

`R`: Numeric

Factorial

For a positive integer `R`: the product of all positive integers through `R`.

```
      !4 3 5
24 6 120
```

`R`: Numeric, excluding negative integers

Find

A Boolean array `Z` that corresponds to `R`. `Z` contains a `1` where pattern `L` begins in the corresponding position of `R`. All other items of `Z` are `0`.

```
      'AB'⍷'ABABAB'
1 0 1 0 1 0
```

Implicit argument: `⎕CT`

First

The first item of `R`. If `R` is empty, the prototype of `R`.

```
      ↑'ME' 'THEE'
ME
```

Floor

For a real R: the largest integer that does not exceed R.

```
      ⌊.4 ¯.3 ¯3.4
0 ¯1 ¯4
```

R: Numeric
Implicit argument: ⎕CT

Format (default)

A simple character array whose appearance is the same as the display of `R`.

```
      M←2 3⍴⍳6
      ⍕M
1 2 3
4 5 6
      ⍴⍕M
2 5
```

Implicit argument: `⎕PP`

Format by Example

A character array containing the data in `R` formatted using format model `L`. `L` includes control characters and decorators. The control characters are:

`0`	Pad zeros to this position
`1`	Float decorator if negative
`2`	Float decorator if nonnegative
`3`	Float decorator
`4`	Do not float nearest decorator
`5`	Normal digit
`6`	Decorator to right ends the field
`7`	Use next decorator to right for scaled form
`8`	Fill empty positions of field with `⎕FC[3]`
`9`	Pad zeros to this position if nonzero
`.`	Decimal point
`,`	Controlled comma

All other characters are decorators and are floated if requested.

```
      ' $53.50'⍕2 2⍴⍳4
  $1.00  $2.00
  $3.00  $4.00
```

`L`: Simple character vector
`R`: Simple real array
Implicit argument: `⎕FC[⍳5]`

Format by Specification

A character array containing data in `R` formatted using column specifications `L`. Each pair of items in `L` corresponds to a column.

The first of each pair sets column width; the second sets display precision and format (positive for conventional or negative for scaled).

A single pair of integers extends the specification to all columns. A single integer `L` is interpreted as `(0,L)`.

```
      5 3 4 0⍕2 2⍴⍳4
1.000   2
3.000   4
```

`L`: Simple integer vector
`R`: Array of depth 2 or less whose items are simple real scalars or vectors
Implicit argument: `⎕FC[4 6]`

Grade Down

A vector of integers that can be used as an index to sort the subarrays along the first axis of R into descending order.

```
      N←23 11 13 31 12
      ⍒N
4 1 3 5 2
      N[⍒N]
31 23 13 12 11
```

R: Simple nonscalar real numeric array
Implicit argument: ⎕IO

Grade Down with Collating Sequence

Same as Grade Down (⍒R) except that the collating sequence used is defined by L.

```
      'ABCDE'⍒'BEAD'
2 4 1 3
```

L: Simple nonempty nonscalar character array
R: Simple nonscalar character array
Implicit argument: ⎕IO

Grade Up

A vector of integers that can be used as an index to sort the subarrays along the first axis of R into ascending order.

```
      N←23 11 13 31 12
      ⍋N
2 5 3 1 4
      N[⍋N]
11 12 13 23 31
```

R: Simple nonscalar real numeric array
Implicit argument: ⎕IO

Grade Up with Collating Sequence

Same as Grade Up (⍋R) except that the collating sequence used is defined by L.

```
      'ABCDE'⍋'BEAD'
3 1 4 2
```

L: Simple nonempty nonscalar character array
R: Simple nonscalar character array
Implicit argument: ⎕IO

Greater Than

A 1 if L is greater than R; a 0 otherwise.

```
      20 30 40>40 30 20
0 0 1
```

L,R: Real numeric
Implicit argument: ⎕CT

Greater Than or Equal to

A `1` if `L` is greater than or equal to `R`; a `0` otherwise.

```
      20 30 40≥40 30 20
0 1 1
```

`L`,`R`: Real numeric
Implicit argument: `⎕CT`

Index

Selects cross sections of R using a list of index arrays L.

```
      ⎕IO←1
      V←2 2.3 ¯5 999 .01
      3⌷V
¯5
      (⊂3 4)⌷V
¯5 999
```

L: Scalar or vector of nonnegative integers of depth no greater than two.
R: Any array.
Implicit argument: ⎕IO

Index of

The position of the first occurrence in `L` of each item in `R`.

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

`L`: Vector
Implicit arguments: `⎕IO`, `⎕CT`

Index with Axis

Selects cross sections of `R` using a list of index arrays `L` that corresponds to axes `X`.

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

`L`: Scalar or vector of nonnegative integers of depth no greater than two.
`R`: Any array.
`X`: Simple scalar or vector; nonnegative integers: `X∊⍳⍴⍴R`
Implicit argument: `⎕IO`

Interval

R consecutive ascending integers beginning with ⎕IO.

```
      ⍳4
1 2 3 4
```

R: Simple scalar or 1-item vector; nonnegative integer less than 2⋆31
Implicit argument: ⎕IO

Laminate

For a fractional `X`: `L` and `R` joined to form a new axis of length 2.

```
      'ONE',[.5]'TWO'
ONE
TWO
```

`X`: Simple nonempty fractional scalar between `¯1+⎕IO` and `⎕IO+(⍴⍴L)⌈⍴⍴R`
Implicit argument: `⎕IO`

See also Catenate with Axis for integral values of `X`.

Less Than

A `1` if `L` is less than `R`; a `0` otherwise.

```
      20 30 40
```

`L`,`R`: Real numeric
Implicit argument: `⎕CT`

Less Than or Equal to

A `1` if `L` is less than or equal to `R`; a `0` otherwise.

```
      20 30 40≤40 30 20
1 1 0
```

`L`,`R`: Real numeric
Implicit argument: `⎕CT`

Logarithm

The base `L` logarithm of `R`.

```
      5 10⍟125 100
3 2
```

`L`,`R`: Numeric; nonzero

Magnitude

Distance between `0` and `R`.

```
      |6 ¯5 3J4
6 5 5
```

`R`: Numeric

Match

A `1` if `L` and `R` are the same in structure and data; a `0` otherwise.

```
      'YES NO'≡'YES','NO'
0
      'YES NO'≡'YES ','NO'
1
```

Implicit argument: `⎕CT`

Matrix Divide

The solution of a system of linear equations of a vector `L` and a nonsingular matrix `R`, or, if `R` has more rows than columns, a least squares approximation.

```
      1 4⌹2 2⍴1 0 0 2
1 2
```

`L`,`R`: Simple numeric arrays of rank 2 or less

Matrix Inverse

The inverse of a nonsingular matrix.

```
      ⌹2 2⍴1 0 0 2
1 0
0 0.5
```

`R`: Simple numeric array of rank 2 or less

Maximum

The larger of `L` and `R`.

```
      4 0.5⌈1 6
4 6
```

`L`,`R`: Real

Member

A Boolean array `Z` that has the same shape as `L`. An item of `Z` is `1` if the corresponding item of `L` can be found anywhere in `R`. An item of `Z` is `0` otherwise.

```
      8 7 6 7 9∊7 8 9 7
1 1 0 1 1
```

Implicit argument: `⎕CT`

Minimum

The smaller of `L` and `R`.

```
      0.4 6⌊1 0.5
0.4 0.5
```

`L`,`R`: Real

Multiply

The product of `L` and `R`.

```
      0 1×3 4
0 4
```

`L`,`R`: Numeric

Nand

A `1` if either `L` or `R` is `0`; `0` if both `L` and `R` are `1`.

```
      0 0 1 1⍲1 0 1 0
1 1 0 1
```

`L`,`R`: Boolean

Natural Logarithm

The logarithm of `R` to the base e.

```
      ⍟1 2
0 0.6931471806
```

`R`: Numeric; nonzero

Negative

Negative `R` if `R` is positive; positive `R` if `R` is negative.

```
      -¯5 1
5 ¯1
```

`R`: Numeric

Nor

A 1 if both L and R are 0; a 0 otherwise.

```
      0 0 1 1⍱1 0 1 0
0 1 0 0
```

L,R: Boolean

Not

A `1` for each item of `R` that is `0`; a `0` for each item of `R` that is `1`.

```
      ~1 0
0 1
```

`R`: Boolean

Not Equal

A `1` if `L` is not equal to `R`; a `0` otherwise.

```
      20 30 40≠40 30 20
1 0 1
```

Implicit argument: `⎕CT`

Or

A 1 if either L or R is 1; 0 if both L and R are 0.

```
      0 0 1 1∨1 0 1 0
1 0 1 1
```

L,R: Boolean

Partition

Partitions `R` into an array of vectors, with break points wherever corresponding items of `L` increase. Where `L=0` the corresponding items of `R` are omitted.

```
      )LOAD 1 DISPLAY
      DISPLAY 1 1 2⊂'ABC'
┌→─────────┐
│ ┌→─┐ ┌→┐ │
│ │AB│ │C│ │
│ └──┘ └─┘ │
└∊─────────┘
      DISPLAY (SAVES≠' ')⊂SAVES←' A STITCH   IN TIME'
┌→─────────────────────────────┐
│ ┌→┐ ┌→─────┐ ┌→─┐ ┌→───┐ │
│ │A│ │STITCH│ │IN│ │TIME│ │
│ └─┘ └──────┘ └──┘ └────┘ │
└∊─────────────────────────────┘
      M←⍕3 3ρ1 10 3.142 2 100 6.283 3 1000 9.425
      DISPLAY M
┌→───────────┐
↓1   10 3.142│
│2  100 6.283│
│3 1000 9.425│
└────────────┘
      DISPLAY (~∧⌿' '=M)⊂M
┌→──────────────────┐
↓ ┌→┐ ┌→───┐ ┌→────┐ │
│ │1│ │  10│ │3.142│ │
│ └─┘ └────┘ └─────┘ │
│ ┌→┐ ┌→───┐ ┌→────┐ │
│ │2│ │ 100│ │6.283│ │
│ └─┘ └────┘ └─────┘ │
│ ┌→┐ ┌→───┐ ┌→────┐ │
│ │3│ │1000│ │9.425│ │
│ └─┘ └────┘ └─────┘ │
└∊──────────────────┘
```

`L`: Simple scalar or vector of nonnegative integers.
`R`: Nonscalar.

When `L=0` the corresponding item does not appear in the result.

Partition with Axis

Partitions `R` into an array of vectors specified by `L` along axis `X`.

```
      )LOAD 1 DISPLAY
      DISPLAY N←4 3ρι12
┌→───────┐
↓ 1  2  3│
│ 4  5  6│
│ 7  8  9│
│10 11 12│
└~───────┘
      DISPLAY 1 0 1 1⊂[1]N
┌→───────────────────────┐
↓ ┌→┐      ┌→┐      ┌→┐   │
│ │1│      │2│      │3│   │
│ └~┘      └~┘      └~┘   │
│ ┌→───┐   ┌→───┐   ┌→───┐│
│ │7 10│   │8 11│   │9 12││
│ └~───┘   └~───┘   └~───┘│
└∊────────────────────────┘
```

`L`: Simple scalar or vector of nonnegative integers.
`R`: Nonscalar.
`X`: Simple scalar or one item vector.
Implicit argument: `⎕IO`

When `L=0` the corresponding item does not appear in the result.

Pick

An item of `R` as specified by the path indices `L`.

```
      R←'ONE' 'TWO'
      2 1⊃R
T
```

`L`: Integer scalar or vector whose depth is 2 or less.
Implicit argument: `⎕IO`

Pi Times

The product of pi and R.

```
      ○1
3.141592654
```

R: Numeric

Power

L raised to the Rth power.

```
      2 10⋆4 ¯2
16 0.01
```

Fractional values are also supported, yielding roots. When ÷R is an even integer there are two real roots. Since APL2 supports complex numbers, there are always multiple possible roots. The result given is the one with the smallest nonnegative angle in the complex plane. (This rule ensures that a positive real root will be returned if there is one.)

```
      16 ⋆ ÷1 2 3 4
16 4 2.5198421 2
      2.5198421 ⋆ 3
16
      16⋆.75  ⍝ 4th root cubed
8
```

L,R: Numeric

Ravel

A vector consisting of the items of `R` taken in row-major order.

```
      R←2 2⍴'ABCD'
      ,R
ABCD
      ⍴,R
4
```

Ravel with Axis

An array whose contiguous subarrays are R, but structured by either combining axes (if X is integer) or forming a new axis of length 1 (if X is a fraction).

```
      ,[2 3]2 2 2⍴⍳8
1 2 3 4
5 6 7 8
      ,[1.1]10 15
10
15
```

X: Simple scalar fraction, simple scalar, vector of nonnegative integers, or empty
Implicit argument: ⎕IO

Reciprocal

`1` divided by `R`.

```
      ÷5 .2
0.2 5
```

`R`: Numeric; nonzero

Reshape

The items of R in an array of shape L.

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

L: Simple scalar or vector; nonnegative integers

Residue

For a real and positive `L`: the remainder of `R` divided by `L`.

```
      0 8|9 15
9 7
```

`L`,`R`: Numeric
Implicit argument: `⎕CT`

Reverse

An array with the items of R reversed along the last axis.

```
      R←3 3⍴'ABCDEFGHI'
      ⌽R
CBA
FED
IHG
```

Reverse with Axis

An array with items of R reversed along the Xth axis.

```
      R←3 3⍴'ABCDEFGHI'
      ⌽[1]R
GHI
DEF
ABC
```

X: Scalar or 1-item vector; nonnegative integer
Implicit argument: ⎕IO

Roll

A random integer from `⍳R`.

```
      ⎕RL←1144108930
      ?10 10 10 10
3 1 7 7
```

`R`: Positive integer
Implicit arguments: `⎕IO`, `⎕RL`

Rotate

An array with items of R rotated |L positions along the last axis. The sign of L determines the direction of rotation.

```
      R←3 4⍴⍳12
      2 1 ¯1⌽R
 3 4  1  2
 6 7  8  5
12 9 10 11
```

L: Simple integer

Rotate with Axis

An array with items of R rotated |L positions along the Xth axis. The sign of L determines the direction of rotation.

```
      R←3 4⍴⍳12
      0 1 ¯1 ¯2 ⌽[1]R
1  6 11  8
5 10  3 12
9  2  7  4
```

L: Simple integer
X: Nonempty simple scalar; nonnegative integer
Implicit argument: ⎕IO

Shape

The length of each axis of `R`.

```
      R←3 4ρι12
      ρR
3 4
```

Subtract

`L` minus `R`.

```
      8 ¯3-6 ¯5
2 2
```

`L`,`R`: Numeric

Take

For a positive scalar `L`: the first `L` items of `R`.
For a negative scalar `L`: the last `L` items of `R`.

```
      2↑1 2 3 4
1 2
      ¯2↑1 2 3 4
3 4
```

`L`: Simple scalar or vector; integer

Take with Axis

For a positive scalar L: the first L items of the Xth axis of R.
For a negative scalar L: the last L items of the Xth axis of R.

(That is, if X is 1, L rows are taken from R; if X is 2, L columns are taken from R.)

```
      H←2 4⍴'ABCDEFGH'
      2↑[2]H
AB
EF
```

L: Simple scalar or vector; integer
R: Nonscalar array
X: Simple scalar or vector; nonnegative integers
Implicit argument: ⎕IO

Transpose

If L selects all axes of R, produces an array similar to R, but with the axes permuted according to L.

If L includes repetitions of axes, produces a diagonal cross section of R.

```
      2 1 3⍉2 2 2⍴⍳8
1 2
5 6
3 4
7 8
```

L: Simple scalar or vector; nonnegative integer

Transpose (reversed axes)

An array similar to R, but with the order of the axes of R reversed.

```
      R←2 3⍴'ABCDEF'
      ⍉R
AD
BE
CF
```

Without

The items in L that do not occur in R.

```
      'DIET'~'TEARS'
DI
```

L: Scalar or vector
Implicit argument: ⎕CT

Primitive Operators

In this section we use the following notation:

`LO`	left operand of the operator
`RO`	right operand of the operator
`X`	axis along which the operator is applied
`L`	left argument of the derived function
`R`	right argument of the derived function

Syntax	**Name**
`LO/R`	Compress
`LO/[X]R`	Compress with Axis
`L LO¨ R`	Each, Deriving Dyadic
`LO¨ R`	Each, Deriving Monadic
`LO\R`	Expand
`LO\[X]R`	Expand with Axis
`L LO.RO R`	Inner Product
`L ∘.RO R`	Outer Product
`LO/R`	Reduce
`L LO/R`	Reduce N-wise
`L LO/[X]R`	Reduce N-wise with Axis
`LO/[X]R`	Reduce with Axis
`LO/R`	Replicate
`LO/[X]R`	Replicate with Axis
`LO\R`	Scan
`LO\[X]R`	Scan with Axis

Compress

Selects subarrays along the last axis under the control of the vector `LO`. This is a special case of Replicate, where `LO` is Boolean.

```
      1 0 0 1 1/'PLEAT'
PAT
```

Compress with Axis

Selects subarrays along the last axis under the control of the vector `LO`. This is a special case of Replicate with Axis, where `LO` is Boolean.

```
      1 0 1/[1]3 2⍴⍳6
1 2
5 6
```

Each, Deriving Dyadic

The result of function `LO` applied between corresponding pairs of items of `L` and `R`.

```
      1 2 3,¨4 5 6
 1 4  2 5  3 6
```

Each, Deriving Monadic

The result of function `LO` applied to each item of `R`.

```
      ρ¨'TOM' 'DICK'
 3  4
```

Expand

Expansion of the last axis of R under control of Boolean vector L.

```
      A←1 0 1 0 0 1
      A\1 2 3
1 0 2 0 0 3
      A\'ABC'
A B  C
```

See also Scan, the same operator symbol with a function as an operand.

Expand with Axis

Expansion of the `X`th axis of `R` under control of Boolean vector `L`.

```
      1 0 1\[1]2 3⍴⍳6
1 2 3
0 0 0
4 5 6
```

Note: Expand along the first axis can also be represented using the `⍀` symbol.

Inner Product

The result of the LO reduction of each row of L, against each column of R, in all combinations with function RO.

```
      R←2 3⍴⍳6
      L←2 2⍴10×⍳4
      L+.×R
 90 120 150
190 260 330
```

Outer Product

Result of function `RO` applied between pairs of items, one from `L` and one from `R`, in all combinations.

```
      (⍳4)∘.×⍳5
1 2  3  4  5
2 4  6  8 10
3 6  9 12 15
4 8 12 16 20
```

Reduce

The result of the expression produced by inserting function LO between adjacent pairs of items along the last axis of R.

```
      +/1 2 3 4 5
15
```

If the last axis of R is empty, LO is not applied. Instead the *identity function* related to it is applied to the prototype of R. Only primitive functions have known identity functions, so use of a defined function as LO with an empty right argument generates DOMAIN ERROR. In general the identity function produces a value which could be used as an argument to the original function so that its result would be the same as its other argument. Thus the identity functions for addition and subtraction produce 0, while the identity functions for multiplication and division produce 1.

The identity functions for Minimum and Maximum are of special interest, since they return the largest positive value and most negative value that can be represented in the machine. These can be used in many contexts as if they were infinity. Note that the identity for Minimum must be positive, and that for Maximum must be negative, to satisfy the definition of an identity.

See also Reduce N-wise, a variant which produces a dyadic derived function, and Replicate, the same operator symbol with an array as an operand.

Reduce N-wise

Same as Reduce, except `L` specifies the number of items along the last axis of `R` to be considered in each application of the derived function `LO/` to the subarrays along the last axis.

```
      2+/⍳6
3 5 7 9 11
      2=/'HELLO'
0 0 1 0
```

Reduce N-wise with Axis

Same as Reduce with axis, except `L` defines the number of items along the `X`th axis to be considered in each application of the function to the subarrays along the `X`th axis.

```
      3+/[1]4 3⍴⍳12
12 15 18
21 24 27
```

Reduce with Axis

Result of the expression produced by inserting function LO between adjacent pairs of items along the Xth axis of R.

```
      +/[1]3 4⍴⍳12
15 18 21 24
```

Note: Reduce along the first axis can also be represented using the ⌿ symbol.

Replicate

Repetition of each subarray along the last axis as specified by array operand `L`.

```
      1 2 3 4/'ABCD'
ABBCCCDDDD
```

See also Reduce, the same operator symbol with a function as an operand.

Replicate with Axis

Repetition of each subarray along the Xth axis as specified by array operand L.

```
      T←3 2⍴⍳6
      2 ¯1 0 1/[1]T
1 2
1 2
0 0
5 6
```

Note: Replicate along the first axis can also be represented using the ⌿ symbol.

Scan

The `I`th item along the last axis is determined by the `LO` reduction of `I↑[ρρR]R`.

```
      +\1 2 3 4 5
1 3 6 10 15
```

See also Expand, the same operator symbol with an array as an operand.

Scan with Axis

The `I`th item along the `X`th axis is determined by the `LO` reduction of `I↑[X]R`.

```
      +\[1]2 3⍴⍳6
1 2 3
5 7 9
```

Note: Scan along the first axis can also be represented using the `⍀` symbol.

System Functions

Syntax	Description and Example
`Z←⎕AF R`	Atomic Function
`Z←L ⎕AT R`	Attributes
`Z←⎕CR R`	Character Representation
`Z←⎕DL R`	Delay
`Z←L ⎕EA R`	Execute Alternate
`Z←⎕EC R`	Execute Controlled
`Z←⎕ES R`	Event Simulation (standard error message)
`Z←L ⎕ES R`	Event Simulation (tailored error message)
`Z←⎕EX R`	Expunge
`Z←⎕FX R`	Fix
`Z←L ⎕FX R`	Fix (with execution properties)
`Z←⎕NA R`	Name Association (Inquire)
`Z←L ⎕NA R`	Name Association (Define)
`Z←⎕NC R`	Name Class
`Z←⎕NL R`	Name List (default)
`Z←L ⎕NL R`	Name List (qualified)
`Z←⎕SVC R`	Shared Variable Control (Inquire)
`Z←L ⎕SVC R`	Shared Variable Control (Set)
`Z←⎕SVO R`	Shared Variable Offer (Inquire)
`Z←L ⎕SVO R`	Shared Variable Offer (Offer)
`Z←⎕SVQ R`	Shared Variable Query
`Z←⎕SVR R`	Shared Variable Retraction
`Z←⎕SVS R`	Shared Variable State
`Z←L ⎕TF R`	Transfer Form
`Z←⎕UCS R`	Universal Character Set

Atomic Function

`Z←⎕AF R`

`⎕AF` converts integers to characters and characters to integers.

Note: `⎕AF` depends on internal representations. Refer to `⎕UCS` for system-independent mapping.

```
      ⎕AF 'A⍳B'
65 236 66
      ⎕AF 65 236 66
A⍳B
```

Attributes

```
Z←L ⎕AT R
```

`⎕AT` returns an attribute vector for each object name specified in `R`. `L` specifies the type of attributes to be returned. The values of `L` and their meanings are:

1	Valence-integer vector (explicit result, function valence, operator valence)
2	Fix time-integer vector (year, month, day, hour, minute, second, millisecond)
3	Execution properties-Boolean vector (nondisplayable, nonsuspendable, ignore attention, reflect domain error)
4	Size of a variable-integer vector (full CDR size, data CDR size)

```
      )LOAD 1 EXAMPLES
SAVED 1993-12-17 09.28.00 (GMT-7)
      ⎕NC 3 4⍴'HOW AND TYPE'
2 4 3
```

This had nothing to do with `⎕AT` directly, but just told us that `HOW` is a variable, `AND` is an operator, and `TYPE` is a function.

```
      1 ⎕AT 3 4⍴'HOW AND TYPE'
1 0 0
1 2 2
1 1 0
```

All three of them have a result. The variable has no valence; the operator is dyadic, producing a dyadic function; and the function is monadic, with no operator valence.

```
      2 ⎕AT 3 4⍴'HOW AND TYPE'
   0 0  0 0 0 0   0
1991 6 29 8 0 0 244
1991 6 29 8 0 0 244
      3 ⎕AT 3 4⍴'HOW AND TYPE'
0 0 0 0
0 0 0 0
0 0 0 0
      4 ⎕AT 3 4⍴'HOW AND TYPE'
8224 8190
   0    0
   0    0
```

Character Representation

`Z←⎕CR R`

`⎕CR` provides a character matrix representation of the defined function or defined operator named in `R`.

An empty matrix is returned for system functions, and for locked user programs or external objects.

```
      )LOAD 1 UTILITY
SAVED 1993-12-17 09.30.46 (GMT-7)
      ⎕CR 'MAT'
A←MAT B
⍝ A is a matrix containing all items of B.
A←,[⍳1+⍴⍴B](1 1,⍴B)⍴B
```

Delay

```
Z←⎕DL R
```

`⎕DL` halts processing for about `R` seconds and returns the actual length of the pause.

```
      ⎕DL 3
3.01
```

Execute Alternate

`Z←L ⎕EA R`

`⎕EA` executes `R`. If `R` fails or is interrupted, `L` is executed.

```
      ∇Z←L DIV R
[1]   Z←'⌊/⍳0' ⎕EA(⍕L),'÷',⍕R
[2]   ∇
      2 DIV 4
0.5
      1 DIV 1
1
      0 DIV 0
1
      1 DIV 0
1.797693135E308
```

In case you are wondering, `⌊/⍳0` produces the largest representable number. Note that the Divide primitive would have done this:

```
      0÷0
1
      1÷0
DOMAIN ERROR
      1÷0
      ^^
```

Execute Controlled

`Z←⎕EC R`

`⎕EC` executes the APL2 expression represented by `R`. The result is a 3-item vector containing:

1. Return code

`0`	Error (`2+'A'`)
`1`	Expression with a result that would display `(2+3)`
`2`	Expression with a result that would not display `(A←2+3)`
`3`	Expression with no explicit result (`F X` where `F` has no result)
`4`	Branch to a line (`→3`)
`5`	Branch escape(`→`)

2. `⎕ET`
3. Result of expression or `⎕EM`

Example:

```
      ⎕EC '2+2'
 1  0 0  4
      ⎕EC 'X←2+2'
 2  0 0  4
      ⎕EC 'X→2+2'
 0  2 2   SYNTAX ERROR
                X→2+2
                 ^
      ⎕EC '→2+2'
 4  0 0  4
```

Event Simulation

`Z←⎕ES R`	Monadic - standard error message
`Z←L ⎕ES R`	Dyadic - tailored error message

The monadic form of `⎕ES` simulates an event and produces an error report for the event based on the value of `R`.

- If `R` is empty or `0 0`, no error is signaled.
- If `R` is a pair of integers, it is assigned to `⎕ET`, and if it is a standard error code, the associated message is used in `⎕EM`.
- If `R` is a character vector, `⎕ET` is set to `0 1`, and `R` is used as the message in `⎕EM`.

The dyadic form does the same thing, but uses `L` as the error message and `R` as the `⎕ET` code.

`⎕ES 0 0` resets the values of `⎕ET` and `⎕EM` to their defaults.

```
      ∇CALL Where
[1]    'Not numeric' ⎕ES(0≠↑0⍴Where)/5 4
[2]    ⎕ES(1≠⍴,Where)/5 3
[3]    ⎕ES(6≠⌊10⍟Where)/'Not a 7-digit number'
[4]    'OK'
[5]    ∇
      CALL 'HOME'
Not numeric
      CALL 'HOME'
      ^
      CALL 123 4567
LENGTH ERROR
      CALL 123 4567
      ^
      CALL 123456
Not a 7-digit number
      CALL 123456
      ^
      CALL 1234567
OK
```

Expunge

`Z←⎕EX R`

`⎕EX` erases object `R` from the workspace. Returns a `1` if the name is available for use; otherwise, returns a `0`.

```
      X∆←1
      ⎕EX 'X∆'
1
      ⎕EX 'X∆'
1
      ⎕EX 'X$'
0
```

Note that `⎕EX` doesn't care whether the name is currently in use, but it does return `0` for something that is an invalid name.

Fix

`Z←⎕FX R`	Monadic - no execution properties
`Z←L ⎕FX R`	Dyadic - with execution properties

`⎕FX` establishes in the active workspace the defined function or operator whose character representation is in `R`, either as a matrix or a vector of vectors.

If the operation succeeds, the name of the program is returned as the explicit result. If `R` is not a valid function or operator definition, a scalar integer is returned indicating the (origin dependent) row or item of `R` where the problem was detected.

For the dyadic form, execution properties are specified by Boolean vector `L`. When the monadic form is used, the execution properties are defaulted to `0 0 0 0`.

The items of `L` and their corresponding execution properties are:

`[1]`	No display
`[2]`	No suspension
`[3]`	Ignore attention
`[4]`	Nonresource error becomes `DOMAIN ERROR`

A `1` in an item of `L` turns the corresponding execution property on; a `0` turns the corresponding execution property off.

```
      ⎕FX 'Z←L PLUS R' 'Z←L+R'
PLUS
      3 PLUS 4
7
      ⎕CR 'PLUS'
Z←L PLUS R
Z←L+R
      1 0 0 0 ⎕FX ⎕CR 'PLUS'
PLUS
      3 PLUS 4
7
      ⎕CR 'PLUS'
      ⍴⎕CR 'PLUS'
0 0
      ⍴⎕CR '+'
0 0
```

Now no one can see how amazingly simple that function is, just as they can't see how the primitive function works.

Name Association

`Z←⎕NA R`	Monadic - Inquire
`Z←L ⎕NA R`	Dyadic - Define

The dyadic form of `⎕NA` associates names `R` with external objects that are accessed through an associated processor. `L` is a 2-item integer vector as follows:

`[1]` Array passed to the processor
`[2]` Associated processor number

The monadic form queries the association of objects named in `R`. It returns the left arguments previously passed to dyadic `⎕NA` when the association was established, or `0 0` if the name does not represent an external object.

For more information about associated processors, see the User's Guide.

```
      ⎕NA 'FILE'
0 0
      3 11 ⎕NA 'FILE'
1
      ⎕NA 'FILE'
3 11
```

Name Class

`Z←⎕NC R`

`⎕NC` returns the name class of each object named in `R`. The meaning of the returned values are:

`¯1`	Invalid
`0`	Unused but valid
`1`	Label
`2`	Variable
`3`	Function
`4`	Operator

```
      )LOAD 1 EXAMPLES
SAVED 1993-12-17 09.28.00 (GMT-7)
      ⎕NC 3 4⍴'HOW AND WHEN'
2 4 0
```

Name List

`Z←⎕NL R`	Monadic - default
`Z←L ⎕NL R`	Dyadic - qualified

`⎕NL` returns a matrix containing names of user-defined objects (labels, variables, functions, and/or operators) in the active workspace whose current name class is a member of `R`. External objects are also included if an association to them is active.

The dyadic form adds additional control by restricting the returned list to names whose first character is in `L`.

```
      )LOAD 1 EXAMPLES
SAVED 1993-12-17 09.28.00 (GMT-7)
      ⎕NL 2   ⍝ List variables
ABSTRACT
CHANGE_ACTIVITY
COIBM
DCS
DESCRIBE
GPAPL2
GPDESC
HOW
TIMER
      ⍴⎕NL 2
9 15
      ⍴⎕NL 4   ⍝ How many operators?
18 7
      'CDE' ⎕NL 4
COMMUTE
CR
EL
ELSE
ER
```

Shared Variable Control

`Z←⎕SVC R`	Monadic - Inquire
`Z←L ⎕SVC R`	Dyadic - Set

The dyadic form of `⎕SVC` sets the Access Control Vector (ACV) for shared variables named in `R` to values in Boolean vector `L`, and returns the effective control, which is an *OR* of the control specified in `L` and the control imposed by the share partner. The ACV is a 4-bit vector indicating:

`1 0 0 0`	My sets to the shared variable are controlled.
`0 1 0 0`	My partner's sets to the shared variable are controlled.
`0 0 1 0`	My references of the shared variable are controlled.
`0 0 0 1`	My partner's references of the shared variable are controlled.

The monadic form of `⎕SVC` returns the access control vectors (ACVs) currently effective control on the variables named in `R`.

```
      100 ⎕SVO 'CMD'
1
      ⎕SVO 'CMD'
2
      ⎕SVC 'CMD'
0 0 0 1
      1 0 1 0 ⎕SVC 'CMD'
1 0 1 1
```

When AP 100 matched the offer, it set the ACV to control its own references, so that it could reliably accept each command passed to it.

Shared Variable Offer

`Z←⎕SVO R`	Monadic - Inquire
`Z←L ⎕SVO R`	Dyadic - Offer

The dyadic form of `⎕SVO` offers to share variables named in `R` with processors named in `L`, and returns the degree of coupling:

`0`	The name cannot be offered as a shared variable.
`1`	An offer has been made, but not yet accepted by the partner.
`2`	The variable is shared with the specified partner.

Note: Most processors are running asynchronously with the interpreter, so the normal response to an initial offer is `1`. The functions `SVOFFER` and `SVOPAIR` in the `UTILITY` workspace are strongly recommended, since they ensure that offers are accepted.

The monadic form of `⎕SVO` checks the current state of a prior offer, returning one of the values listed above. (`0` is returned if the variable has not been offered.)

If the dyadic form is used with a variable that was previously offered, and if the processor number matches the previous offer, then the function behaves like the monadic form.

```
      100 ⎕SVO 'CMD'
1
      ⎕SVO 'CMD'
2
      100 ⎕SVO 'CMD'
2
      101 ⎕SVO 'CMD'
0
```

The third `⎕SVO` was treated as if it was a query, but the fourth one responded that an offer could not be made since that name had already been offered to a different processor.

Shared Variable Query

`Z←⎕SVQ R`

If `R` is an empty vector, `⎕SVQ` returns a numeric vector of processors making share offers to you. If `R` contains one or more integers, the result is a matrix of names of variables being offered by the processors listed in `R`.

Session A / **Session B**

```
Session A                             Session B

    ↑⎕AI ⍝ Who am I?
1001


                                          ↑⎕AI
                                      1002


    1002 ⎕SVO 'TALK'
1


                                            ⎕SVQ ⍳0
                                      1001
                                            ⎕SVQ 1001
                                      TALK
```

Shared Variable Retraction

```
Z←⎕SVR R
```

⎕SVR requests retraction of the shared variable or variables named in R and returns their prior degree of coupling. For the meanings of the degree of coupling values, see Shared Variable Offer.

```
      100 ⎕SVO 'CMD'
1
      ⎕SVR 'CMD'
2
```

Between the time that the offer was made and the time it was retracted, AP 100 had matched the offer, thus explaining the response of 2.

Shared Variable State

```
Z←⎕SVS R
```

`⎕SVS` returns the Access State Vector (ASV) of each variable named in `R`. The ASV is a 4-item Boolean vector containing one of the following three combinations of values:

`1 0 1 0`	You have set a value that your partner has not yet referenced.
`0 1 0 1`	Your partner has set a value that you have not yet referenced.
`0 0 1 1`	Both partners have seen the current value, or no value has been set since the variable was shared.

```
      1234 ⎕SVO 'TEST'
1
      TEST←'HELLO'
      ⎕SVS 'TEST'
1 0 1 0
```
In this case no one matched the share offer, so the (missing) partner will never reference the value.

Transfer Form

`Z←L ⎕TF R`

If `R` is a valid APL name, `⎕TF` returns the transfer form as specified in `L` of the variable or displayable defined functions or defined operators named in `R`. `2 ⎕TF` also supports associations to external objects.

If `R` is a valid transfer form, of the type specified in `L`, the object is established in the active workspace, and its name is returned.

`L` must be either `1` for migration form or `2` for extended form. Migration form is intended for compatibility with systems without the data extensions in APL2. It does not support external objects, mixed or nested arrays, or complex numbers.

```
      NUMS←3 8 2
      MIX←'APL' 2
      2 ⎕TF 'NUMS'
NUMS←3 8 2
      2 ⎕TF 'MIX'
MIX←('APL')(2)
      1 ⎕TF 'NUMS'
NNUMS 1 3 3 8 2
      1 ⎕TF 'MIX'
      )CLEAR
CLEAR WS
      2 ⎕TF 'NUMS←3 8 2'
NUMS
      NUMS
3 8 2
```

Universal Character Set

```
Z←⎕UCS R
```

`⎕UCS` converts integers to characters and characters to integers using the ISO 10646 standard, which includes the Unicode subset. `⎕UCS` supports the 2-byte Unicode character set.

For character arguments, `⎕UCS` returns a numeric result containing the Unicode code points for the characters. For extended characters not in the 2-byte Unicode character set, it returns the numeric values stored in the characters, unchanged.

For numeric arguments, `⎕UCS` returns a character result based on the Unicode code points in the argument. If the Unicode character represented by a code point is not in the APL2 character set, it returns an extended character with the value from the argument, unchanged.

```
      ⎕UCS 'A⍳B'
65 9075 66
      ⎕UCS 65 9075 66
A⍳B
```

Contrast this with `⎕AF`, which returns system-dependent results. For ASCII-based systems,

```
      ⎕AF 'A⍳B'
65 236 66
```

For EBCDIC-based systems,

```
      ⎕AF 'A⍳B'
193 178 194
```

System Variables

Syntax	Description and Example
`⎕AI`	Account Information
`⎕AV`	Atomic Vector
`⎕CT←X`	Comparison Tolerance
`⎕EM`	Event Message
`⎕ET`	Event Type
`⎕FC←X`	Format Control
`⎕IO←X`	Index Origin
`⎕LC`	Line Counter
`⎕LX←X`	Latent Expression
`⎕NLT`	National Language Translation
`⎕PP←X`	Printing Precision
`⎕PR←X`	Prompt Replacement
`⎕PW←X`	Printing Width
`⎕RL←X`	Random Link
`⎕SVE←X`	Shared Variable Event
`X←⎕SVE`	
`⎕TC`	Terminal Control
`⎕TS`	Time Stamp
`⎕TZ←X`	Time Zone
`⎕UL`	User Load
`⎕WA`	Workspace Available
`X←⎕`	Evaluated Input/Output
`⎕←X`	
`⍞←X`	Character Input/Output
`X←⍞`	

Account Information

`⎕AI`

The meanings of the `⎕AI` items are:

`[1]` Account number (also shared variable processor ID)
`[2]` Compute time
`[3]` Connect time
`[4]` Keying time

Times are in milliseconds.

```
      ⎕AI
1001 70870 30097000 30026130
```

Atomic Vector

`⎕AV`

The 256 possible single-byte characters, sorted in sequence.

```
      16 16⍴⎕AV
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
ÇüéâäàåçêëèïîìÄÅ
⎕⍞⌹ôöòûù⊤ÖÜø£⊥₧⌶
áíóúñÑªº¿⌈¬½∪¡⍕⍎
░▒▓│┤⍟∆∇→╣║╗╝←⌊┐
└┴┬├─┼↑↓╚╔╩╦╠═╬≡
⍸⍷∵⌷⍂⌻⊢⊣⋄┘┌█▄¦Ì▀
αß⊂⊃⍝⍲⍴⍱⌽⊖○∨⍳⍉∊∩
⌿⍀≥≤≠×÷⍙∘⍵⍫⍋⍒¯¨
```

The character set represented by `⎕AV` is system-dependent. On workstation systems, `⎕AV` contains the APL2 ASCII characters, as defined by IBM code page 910. On mainframe systems, `⎕AV` contains the APL2 EBCDIC characters, as defined by IBM code page 293.

Note that the first 32 positions of the ASCII character set contain control characters, so the first two lines of the above display are shown here as blanks. Use of the characters in those positions may cause side effects such as unexpected line breaks.

In addition to being different across systems, quite a number of the characters in the APL2 code pages are different from those defined by various national language code pages. For system-independent mapping of characters, or to use characters not in `⎕AV`, see the `⎕UCS` system function, which uses Unicode code points.

Comparison Tolerance

`⎕CT←X`

A value used by some primitive functions to determine whether two numbers can be considered equal; the default is `1E¯13`.

```
      2=2.00000000000001
1
      ⎕CT←0
      2=2.00000000000001
0
```

Note: No number is within `⎕CT` of `0`.

Event Message

```
⎕EM
```

Text of the error message associated with the first (i.e. current) line of the state indicator.

```
      ⎕EM
      ρ⎕EM
3 0
      2+'A'
DOMAIN ERROR
      2+'A'
      ^^
      ⎕EM
DOMAIN ERROR
      2+'A'
      ^^
      ρ⎕EM
3 12
      →0
      ρ⎕EM
3 0
```

Event Type

`⎕ET`

A 2-integer code that indicates the type of error associated with the first line of the state indicator. Event type codes fall into several major classes. The codes and their meanings are summarized by each major class in the following lists.

Note: `⎕ET` is also set by the `⎕ES` system function. In this case its values may be application defined.

Defaults:

`0 0`	No error
`0 1`	Unclassified event

Resource Errors:

`1 1`	`INTERRUPT`
`1 2`	`SYSTEM ERROR`
`1 3`	`WS FULL`
`1 4`	`SYSTEM LIMIT` - symbol table
`1 5`	`SYSTEM LIMIT` - interface unavailable
`1 6`	`SYSTEM LIMIT` - interface quota
`1 7`	`SYSTEM LIMIT` - interface capacity
`1 8`	`SYSTEM LIMIT` - array rank
`1 9`	`SYSTEM LIMIT` - array size
`1 10`	`SYSTEM LIMIT` - array or internal function depth
`1 11`	`SYSTEM LIMIT` - prompt length
`1 12`	`SYSTEM LIMIT` - value unrepresentable
`1 13`	`SYSTEM LIMIT` - implementation restriction

Syntax Error:

`2 1`	Required operand or right argument omitted
`2 2`	Ill-formed line
`2 3`	Name class
`2 4`	Invalid operation in context
`2 5`	Compatibility setting prohibits this syntax

Value Error:

`3 1`	Name with no value

3 2	Function with no result

Implicit Argument Errors:

4 1	⎕PP ERROR
4 2	⎕IO ERROR
4 3	⎕CT ERROR
4 4	⎕FC ERROR
4 5	⎕RL ERROR
4 7	⎕PR ERROR

Explicit Argument Errors:

5 1	VALENCE ERROR
5 2	RANK ERROR
5 3	LENGTH ERROR
5 4	DOMAIN ERROR
5 5	INDEX ERROR
5 6	AXIS ERROR

```
      )CLEAR
CLEAR WS
      ⎕ET
0 0
      2+'A'
DOMAIN ERROR
      2+'A'
      ^^
      ⎕ET
5 4
```

Format Control

⎕FC←X

Characters used for the Format function (L⍕X). See Format by Example and Format by Specification. The meanings of the ⎕FC items are:

[1] Decimal character
[2] Thousands indicator
[3] Fill for blanks
[4] Fill for overflow
[5] Print-as-blank
[6] Negative indicator

Starting with a default ⎕FC:

```
      ⎕FC
.,⋆0_¯
      ' $888,888.00'⍕1234.5
 $⋆⋆1,234.50
```

Now let's use the European convention for comma and period, and change the fill character.

```
      ⎕FC←',.=0_¯'
      ' $888,888.00'⍕1234.5
 $==1.234,50
```

Finally, here is something we can do about overflows:

```
      ' 5550'⍕2 1⍴123 45678
DOMAIN ERROR
      ' 5550'⍕2 1⍴123 45678
      ^      ^
      ⎕FC←',.=⋆_¯'
      ' 5550'⍕2 1⍴123 45678
  123
 ⋆⋆⋆⋆
```

Index Origin

`⎕IO←X`

Index of the first item of a non-empty vector.

```
     ⎕IO   ⍝ The default
1
      'ABCD'[0]
INDEX ERROR
      'ABCD'[0]
      ^
      'ABCD'[2]
B
      ⎕IO←0
      'ABCD'[2]
C
      ⎕IO←2
      'ABCD'[2]
⎕IO ERROR
      'ABCD'[2]
      ^
```

Only `0` or `1` are valid values.

Line Counter

⎕LC

Line numbers of defined functions or operators that are in execution or halted; the most recently activated line number is first.

```
      )SI
LOG[6]
CMD[7]
CMVC_VIEW[4]
DIALOG1[9]
CANSWER[21]
⋆
AP124[9]
DIALOG1[33]
CANSWER[21]
⋆
      ⎕LC
6 7 4 9 21 9 33 21
```

Note that immediate execution levels, marked with ⋆ in the)SI list above, are not included in the ⎕LC vector.

Latent Expression

`⎕LX←X`

`⎕LX` is an APL2 statement to be executed automatically by the Execute function (`⍎`) when the workspace is loaded. The default is `''`, which in effect means to do nothing, and prompt for input.

National Language Translation

`⎕NLT`

National language in which system messages are displayed and system commands can be entered.

Note: `⎕NLT` is not fully implemented on the workstation APL2 systems. The variable cannot be set to effect, and always reports the name of the file currently in use for messages. The `-nlt` invocation option can be used to specify the name of the message file.

Printing Precision

`⎕PP←X`

The number of significant digits displayed for numbers.

```
      ⎕PP   ⍝ The default
10
      ○1    ⍝ Value of pi
3.141592654
      ⎕PP←4
      ○1
3.142
```
Note that values are automatically rounded to the precision.

```
      ⎕PP←15
      ○1
3.14159265358979
      ⎕PP←20
      ○1
3.141592653589793
```
The maximum supported value is `16`; any larger values are treated as if that had been specified.

Prompt Replacement

`⎕PR←X`

A character that replaces the prompt string during character (⍞) input.

```
      ∇Z←NAME
[1]   ⍞←'Enter your name: '
[2]   Z←⍞∇
      ⎕PR   ⍝ Default is blank
      NAME
Enter your name: Bob
                 Bob
      ⎕PR←'⋆'
      NAME
Enter your name: Bob
⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆Bob
      ⎕PR←''
      NAME
Enter your name: Bob
Enter your name: Bob
```

Note that in the last case `⎕PR` was set to an empty vector, indicating no replacement, rather than the default single blank.

Printing Width

`⎕PW←X`

The maximum number of characters per line of output.

```
      ⎕PW   ⍝ The default
79
      79⍴'123456789-'
123456789-123456789-123456789-123456789-123456789-123456789-123456789-123456789
      ⎕PW←30  ⍝ The minimum
      79⍴'123456789-'
123456789-123456789-123456789-
      123456789-123456789-1234
      56789-123456789-12345678
      9
```

Random Link

```
⎕RL←X
```

Base number for computing random numbers.

```
      ⎕RL  ⍝ Initial CLEAR WS value
16807
      ?6   ⍝ Roll a die
1
      ?6
5
      ⎕RL
1622650073
      ⎕RL←16807   ⍝ Restore default
      ?6 6        ⍝ Roll 2 dice
1 5
```

Valid values are positive integer scalars less than `¯1+2⋆31`.

Shared Variable Event

```
⎕SVE←X
X←⎕SVE
```

On assignment: Specifies the amount of time in seconds to be used in a wait for a shared variable event and starts the timer.

On use: Suspends execution until the specified number of seconds has elapsed or a shared variable event occurs. When an event occurs, returns the time remaining in the timer. Will return immediately if an unhandled event has already occurred. Completion of reference clears any record of events to that point.

```
      ∇SERVE[⎕]
[0]   Z←Name SERVE Value
[1]   ⍝ Assigns "Value" to "Name",
[2]   ⍝ then waits up to 30 seconds
[3]   ⍝ for any partner request.
[4]   ⍝ Returns length of time waited.
[5]   ⍝
[6]    ⎕SVE←0  ⍝ Reset previous signals
[7]    Z←⎕SVE  ⍝ Finishes immediately
[8]    ⍎Name,'←Value'
[9]    ⎕SVE←30  ⍝ Maximum wait
[10]   Z←30-⎕SVE
```

Terminal Control

`⎕TC`

Terminal control characters. The items of `⎕TC` are:

`[1]` Backspace
`[2]` New line (or Carriage Return)
`[3]` Line feed

Note that `¯2↑⎕TC` is the normal PC convention for end of record in files.

Time Stamp

⎕TS

Current system date and time. The items of ⎕TS are:

[1]	Year
[2]	Month
[3]	Day
[4]	Hour
[5]	Minute
[6]	Second
[7]	Millisecond

All values are integers.

Time Zone

```
⎕TZ←X
```

Offset in hours between local time and Greenwich Mean Time; the values are negative for locations west of GMT.

```
      ⎕TZ
¯7
      3 11 ⎕NA 'GETENV'
1
      GETENV ⊂'TZ'
 pst8pdt
```

A simple form of TZ is being used here, that assumes normal US rules for the beginning and end of daylight savings time. The result of `⎕TZ` was `¯7` rather than `¯8` because daylight savings time was in effect.

User Load

```
⎕UL
```

Number of users.

```
      ⎕UL
1
```

On PC systems, you always get 1, since they are single user systems. On Unix and mainframe systems, you will get the number of users logged on to the operating system.

Workspace Available

`⎕WA`

Number of available (unused) bytes in the active workspace.

```
CLEAR WS
      ⎕WA
2090096
      2×1024⋆2
2097152
```

Note that the default for the -ws invocation option is 4M, which really means `4×1024⋆2`. The amount actually available is always somewhat less than that, because of internal tables.

Evaluated Input/Output

```
X←⎕
⎕←X
```

On assignment, displays the value and starts a new line in the session input/output stream.

On reference, displays `⎕:` to prompt for input, evaluates the expression entered in response, and uses that result as the value of `⎕`.

```
      ∇Z←QUAD
[1]   ⎕←'First message'
[2]   ⎕←'Second message'
[3]   Z←⎕∇
      QUAD
First message
Second message
⎕:
      2+2
4
```

Character Input/Output

```
⍞←X
X←⍞
```

On assignment, displays the value and leaves the cursor on the same line. (Note that ⎕TC characters may be imbedded in the value. They will be honored if so, and can cause a single vector to be displayed on multiple lines, or the cursor to be positioned somewhere other than the last character.)

On reference, obtains raw input from the keyboard (or AP 101 stack). Frequently such input follows a ⍞ output request. In this case the output prompt data on the same line occupies space in the result of ⍞, but has been converted as specified by ⎕PR.

```
      ∇Z←QQUAD
[1]    ⍞←'First '
[2]    ⍞←'Second '
[3]    Z←⍞∇
      QQUAD
First Second 2+2
              2+2
```

System Commands

System command keywords can be entered in any combination of uppercase and lowercase letters.

Braces `{ }`, brackets `[ ]`, elipses `...`, and the `|` symbol are used here as *metasyntax*, indicating lists of options, optional data, repeated fields, and alternatives respectively. They should never appear in the system commands you actually enter.

The following symbols are used, in lower case, throughout the list of system commands. They should be replaced by usage-specific values as indicated:

Symbol	**Definition**
`ext`	file extension
`file`	file name
`libno`	A positive integer, 1 to 32767, associated with an `APL`*nnnnn* library definition.
`name`	Any valid APL name
`path`	Any valid operating system directory specification, such as `C:\MY\FOLDER` or `/u/userid/folder`.
`wsname`	1 to 64 alphanumeric characters, the first alphabetic.

The following list shows the syntax of each system command, and provides a brief comment on its usage. Full details are provided in *APL2 Programming: Language Reference*.

```
)CHECK WS [OFF|ON|ALL|SLOP]
)CHECK TRACE {OFF|SERVICE|STMT|EXEC|SYNT|FREE|NS}
)CHECK DUMP
```

Obtain diagnostic information.
The `)CHECK WS` commands control internal validation of the workspace.
The `)CHECK TRACE` commands control tracing of internal interpreter events.
`)CHECK DUMP` causes an immediate SYSTEM ERROR.
Note: The `)CHECK` commands can create large amounts of output and significantly degrade performance. They are best used under the direction of your IBM support personnel.

```
)CLEAR
```

Clear the active workspace.

```
)CONTINUE
```

Save the active workspace as `CONTINUE` and end the APL2 session.

```
)COPY {'[path\]file.ext' | [libno] wsname} [name|(name) ...]
```

Copy all or specified objects into the active workspace from a saved workspace.
If `name` is enclosed in parentheses, it must be a simple character scalar, vector, or matrix. Its rows are interpreted as APL names, and these objects are copied instead of the array itself.

```
)DROP {'[path\]file.ext' | [libno] wsname}
```

Delete a workspace file or transfer file.

```
)EDITOR [1 | editorname [-u]]
```

Query the active editor, or select an editor. Editor 1 is the line editor.

```
)ERASE name|(name) [name|(name) ...]
```

Delete objects from the active workspace.
If `name` is enclosed in parentheses, it must be a simple character scalar, vector, or matrix. Its rows are interpreted as APL names, and these objects are erased instead of the array itself.

`)FNS [[name | name-name] ...]`
List names of defined functions in the active workspace.

`)HOST [command]`
Query the name of the operating system, or issue an operating system command from the APL2 session.

`)IN {'[path\]file.ext' | [libno] wsname} [name ...]`
Read objects from a transfer file into the active workspace.

`)LIB ['path' | libno] [[name | name-name] ...]`
List workspace names in a library or directory.

`)LOAD {'[path\]file.ext' | [libno] wsname}`
Bring a workspace from a library into the active workspace.

`)NMS [[name | name-name] ...]`
List names in the active workspace. See also `)FNS`, `)OPS`, and `)VARS`.

`)OFF`
End the APL2 session.

`)OPS [[name | name-name] ...]`
List names of defined operators in the active workspace.

`)OUT {'[path\]file.ext' | [libno] wsname} [name ...]`
Write objects to a transfer file. (Replaces any existing fileid.)

`)PCOPY {'[path\]file.ext' | [libno] wsname} [name|(name) ...]`
Copy objects into the active workspace without replacing existing objects.
If `name` is enclosed in parentheses, it must be a simple character scalar, vector, or matrix. Its rows are interpreted as APL names, and these objects are copied instead of the array itself.

`)PIN {'[path\]file.ext' | [libno] wsname} [name ...]`
Read objects from a transfer file into the active workspace without replacing existing objects.

`)RESET [number]`
Clear all or a portion of the state indicator - same as `)SIC`.

`)SAVE ['[path\]file.ext' | [libno] wsname]`
Save the active workspace.

`)SI [number]`
Display all or part of the state indicator.

`)SIC [number]`
Clear all or the specified number of levels of the state indicator.

`)SINL [number]`
Display the state indicator and the local variables at each level.

`)SIS [number]`
Display the state indicator and the current statement at each level.

`)SYMBOLS [number]`
Query or modify the symbol table size.

`)VARS [[name | name-name] ...]`
List the names of user variables in the active workspace.

`)WSID ['[path\]file.ext' | [libno] wsname]`
Query or assign the workspace identifier.

Defined Functions and Operators

Many problems can be solved by merely entering APL2 expressions in immediate execution mode. However, when a series of expressions needs to be entered repeatedly in different situations, when a general solution can be applied to several similar problems, or when expressions should be executed based on certain conditions, you may prefer to *define* an operation (a function or operator) to hold the necessary code.

A defined function or operator is *fixed* or established in the active workspace in one of the following ways:

- Created or modified using an editor (see the APL2 User's Guide)
- Created or replaced using ⎕FX or ⎕TF
- Copied, using)COPY or)PCOPY
- Retrieved from a transfer file using)IN or)PIN

(System Commands shows the syntax of the four commands in the last two points above.)

When a defined function or operator is *invoked*, the statements in it are executed. The syntax and order of evaluation when invoking defined functions and operators is the same as for primitive functions and operators.

The definition of a function or operator begins with a *header line* followed by one or more APL statement lines, each of which is in the same format used when entering an APL expression directly. Here is an example of a defined function which rounds a number to a specified number of decimal places. If no number of places is given, 2 is assumed.

```
Z←Y ROUND X
⍝ Round value X to Y decimal places
⍝ Default Y to 2 if not specified
→(0≠⎕NC 'Y')/RN
Y←2            ⍝ Default for Y
RN:Z←(10⋆-Y)×⌊.5+X×10⋆Y
```

Let's try it out:

```
      4 ROUND ○1   ⍝ Pi to 4 digits
3.1416
      ROUND ○1
3.14
```

Although we wrote this thinking about rounding one number, it also works for an array of any shape, rank, or depth,

```
      ROUND 3 7.286
3 7.29
      ROUND 2 2⍴(⍳4)÷7
0.14 0.29
0.43 0.57
      ROUND (1 (2 3 (4 5)))÷7
 0.14   0.29 0.43  0.57 0.71
```

Details of defined functions and operators are covered in the following sections:

- Operation Header
- Branching
- Execution
- Debug Controls

Operation Header

The operation *header* is the first line of a defined operation. The header establishes the syntax for the defined operation, including:

- Name of the operation
- Valence of the operation, and in the case of defined operators, also the valence of the derived function
- Parameter names
- Nature of the result - explicit or not explicit
- Local names

These topics are covered in the following sections:

- Nature of the result
- Defined Function Valence
- Defined Operator Valence
- Local Names

The name of the operation and its parameter names are discussed with function and operator valences.

Nature of the result

If the operation has (or may have) an explicit result, the first thing shown on the header line is a name by which the result will be referred to within the operation, followed by a specification arrow. Although this follows the form of an APL specification statement, no result value is assigned by the header itself. A result is returned only if some statement within the operation is evaluated and causes a value to be assigned to the result. Note that the *name* of the result is local to the operation. (See Local Names.)

Defined Function Valence

A defined function can have two, one, or no arguments. The name of the function is shown with parameters on either side where arguments are permitted. Thus (ignoring the optional result) there are three valid forms:

```
la name ra
   name ra
   name
```

The `la` and `ra` fields indicate whether a left argument or right argument respectively can be provided to the function. The values shown on the header line are parameter names which are local to the operation. (See Local Names.) These are ordinary array names within the function, and their values can be modified if desired. Changes to the values do not affect the data values used by the caller when invoking the function.

If two arguments are shown in a defined function, the function is *ambi-valent*, which means that it can be invoked with either one or two arguments. The function definition should check for the possibility that it was called with only a right argument. If not, a `VALUE ERROR` will occur when the function tries to use the missing argument.

Defined Operator Valence

A defined operator can have two or one operands, and its derived function can have two or one arguments. These valences are shown by listing operand parameters inside a pair of parentheses, with argument parameters outside the parentheses.

Since an operator must always have at least one operand, there will always be either two or three names inside the parentheses. Since the left operand is required, the first name will always be a parameter (operand) name, and the second will always be the name of the operator.

The derived function must always take at least one argument, so there will always be a parameter to the right of the closing parenthesis. Here are the valid formats (excluding the optional result):

```
   (lo mop) ra
   la (lo mop) ra
   (lo dop ro) ra
   la (lo dop ro) ra
```

The `la` and `ra` fields are the arguments to the derived function, and follow the same rules as was described for those fields under Defined Function Valence.

The `lo` and `ro` fields contain operand parameter names which are local to the operation. (See Local Names.) These names can be (in fact must be) used exactly as the operands used in invoking the operator would be used. If a function is passed as an operand, then the parameter must be used as a function of the proper valence. It cannot be inspected as if it were a variable. Note that `⎕NC` can be used within the operator definition to determine whether a function or array was passed. This is permitted even if a constant or a primitive or derived function was passed, cases that `⎕NC` treats as invalid when specified directly.

```
      ⎕CR 'TEST'
Z←(LEFT TEST RIGHT)X
Z←(⎕NC 'LEFT')(⎕NC 'RIGHT')
      +TEST×3
3 3
      4 TEST×3
2 3
      +/TEST 7 3
3 2
      ⎕NC '+'
¯1
      ⎕NC '+/'
¯1
      ⎕NC '7'
¯1
```

If an operand is an array, its value can be modified by assignment within the defined operator. If it is a function, its definition can be replaced using `⎕FX` or `⎕TF`. Either way, changes to the operand within the operator do not affect the function or array used by the caller when invoking the operator.

Local Names

The argument, operand, and result parameters have value only within the context of the defined operation. When execution of the operation is completed, the parameter names are no longer associated with the values they had during execution. Thus, they are called *local names* because their values are local to (exist only during execution of) the defined operation.

In addition to parameter names, you can declare other constructed names to be local to the operation. These can be names that hold intermediate values, set system variables especially for the operation, or are otherwise not needed after the operation has been executed.

Such names are identified by listing them on the header line after the valence and parameter definition, separated from it and from each other by semicolons. Note that system variables can also be listed as local names. This allows them to be set within the operation without affecting the value they have outside of it.

Statement labels (used for branch targets) are also local names. They do not need to be listed in the header.

Local names are not completely private to the operation which defines them as local. *Localizing* a name really only separates it temporarily from any value it had before the operation was invoked. If one defined function localizes a name and assigns it a new value, then calls a second function, the second function will see the local value.

```
      ⎕CR'SAY_V'
SAY_V
'V contains' V
      V←3
      SAY_V
 V contains 3
      ⎕CR'TEST'
TEST;V
V←2.9
SAY_V
      TEST
 V contains 2.9
      SAY_V
 V contains 3
```

Branching and Labels

A branch expression explicitly determines the next line of a defined function or operator to be executed. It consists of a branch arrow (→) and an expression:

```
       →expression
```

The expression must evaluate to a number, or to a vector of zero or more numbers. The numbers refer to lines in your program (function or operator). But you should never "hard code" references to a specific line, since it is very likely that at some later time you or someone else may modify the program in a way that causes lines to be added or removed. Instead you should always use labels on the lines you need to refer to, and then use those labels as if they were numeric variables in your branch expressions. Labels are local names, even though they are not listed in the program header. Here is an example of a function which uses a branch label:

```
Z←Y ROUND X
⍝ Round value X to Y decimal places
⍝ Default Y to 2 if not specified
→(0≠⎕NC 'Y')/RN
Y←2            ⍝ Default for Y
RN:Z←(10⋆-Y)×⌊.5+X×10⋆Y
```

Right now, RN evaluates as 5 (since the header counts as line 0), but if you added or deleted a comment earlier in the function that would change.

Note: The behavior of the statement using RN is explained in Branching Example.

The possible branch actions depend on the value of the branch expression:

If the Branch Expression is...	Then the Next Action Is...
Line number n within the program	Line n of the program is evaluated.
0 or any other line number not within the program	The program ends, and evaluation of the expression that invoked it is resumed.
Empty vector	Next sequential expression (either the next expression to the right of a diamond in the same line, or else the next line, if there is one, of the program).
Vector of numbers	The first number determines the branch action.

For further details, see:

- Conditional Branch
- Branch to Escape
- Branch in a Line with Diamonds
- Looping Is Rarely Needed

Branching is also used in immediate execution to resume execution of a suspended immediate execution statement, defined function, or defined operator. This use of branching is discussed in Resume or Restart Execution.

Explanation of Branching Example

```
Z←Y ROUND X
⍝ Round value X to Y decimal places
⍝ Default Y to 2 if not specified
→(0≠⎕NC 'Y')/RN
Y←2              ⍝ Default for Y
RN:Z←(10⋆-Y)×⌊.5+X×10⋆Y
```

This function contains the branch statement:

```
→(0≠⎕NC 'Y')/RN
```

which APL2 evaluates as follows:

- `⎕NC 'Y'` will return `2` if a value has been associated with parameter `Y`; or `0` if no value has been associated with it. This is a test for whether the optional left argument was provided.
- `0≠` will convert this to `0` if there is a left argument, or `1` if there is none.
- `1/RN` will return 1 copy of the value `5`, and the branch will be taken; while `0/RN` will return an empty result, so evaluation will continue with the next sequential statement.

See Conditional Branch for more general discussion and guidance.

Conditional Branch

When a branch expression takes different values depending on relationships or conditions, the branch is called a *conditional branch*. It is constructed by using `→` with relational and selection operations.

The statement `→(0≠⎕NC 'Y')/RN` is a conditional branch statement because its value may be `RN` or the empty vector, depending on the value of the relationship in parentheses.

Conditions for branch expressions evaluate to `0` or `1`. The relational functions (`< ≤ = ≥ > ≠`) are often used to express simple conditions.

Below are three frequently used conditional branch expressions. In each case, the *condition* evaluates to `0` or `1`.

Form	Description
`→(condition)/0`	Exit from the function or operator if the condition is true. Continue with the next sequential expression if the condition is false.
`→(condition...)/label...`	Begin evaluation at the labeled line if the condition is true. Continue with the next sequential expression if the condition is false.

Form	Description
	Any number of conditional expressions can be used as long as there are the same number of labels. If more than one condition is true, the first true one applies.
`→label×condition`	Begin evaluation at the labeled line if the condition is true. Exit from the function or operator if the condition is false.

Note: Compression is the most commonly used operation in constructing branch expressions. It works equally well for a one- or several-way branch. It is not origin dependent. Reshape or Take could also be used, but are seen much less frequently.

Branch to Escape

A branch arrow with no expression on the right causes the defined operation to immediately terminate. The function or operator ends without providing a result, as do all functions or operators in its calling chain. The expression which invoked the first function or operator in the chain is also aborted, and the system prompts for user input. See `⎕EC` for an exception.

Note that, unlike the `)SIC` command, this does not clear all levels of the execution stack. If a program is suspended, and a new statement is entered which invokes a function, leading ultimately to a → without an expression, the system state will return to the point at which the statement was entered, and the originally suspended program will remain on the execution stack.

Branch in a Line with Diamonds

When a branch expression is one of several expressions separated by diamonds, evaluation continues with the expression to the right of the branch expression if the branch is not taken. This can be useful for *if* constructs:

```
→(~condition)/1+⎕LC ◇ expression
```

or *until* constructs:

```
expression ◇ →(~condition)/⎕LC
```

Looping Is Rarely Needed

Many programmers who come to APL2 after using other languages structure their function and operator definitions with the equivalent of DO loops, working with data an item at a time. This approach should be avoided:

- Looping forces APL2 to interpret each expression in the loop each time it is evaluated. This makes your APL programs run much more slowly than they should.
- Looping forces you to maintain control variables, and provide logic that increments them and checks for loop termination. APL already has all that logic built in, and it is almost certainly more efficient and more trouble free than any you could provide.

APL2's array processing and operators help you avoid most looping.

- Scalar functions are entirely data-driven. They allow computations to be performed where the data itself controls the limits of the operation. You can do arithmetic on entire collections of numbers in a single operation. Because so many of APL2's functions are independent of data structure, it is also true in many cases that defined functions, even those written to handle one data item at a time, will automatically extend to handle arbitrary arrays.
- Structural functions allow you to rearrange arrays in a wide variety of ways. You can easily reshape an array to an equivalent one with a different number of rows or axes, so that you can apply operations to particular groupings of data. You can reverse items, transpose axes, rotate individual rows or columns by varying amounts, and much more.
- The Axis operator, described as a *with axis* variant of the primitive functions and operators to which it applies, allows you to apply operations along particular axes.
- The Reduce and Scan operators allow you to apply arbitrary functions between data items in the array.
- The Boolean arrays produced by comparisons, together with the complete set of Boolean functions and the Replicate and Expand operators, allow you to focus on the subset of the data which satisfies any arbitrarily complex set of conditions.
- The Each operator (see Each, Deriving Monadic and Each, Deriving Dyadic) is, in a practical sense, the equivalent of a DO loop, except that the loop limits are implicit in the data.

Execution

When the name of a defined operation appears in an expression, its context is evaluated following the same evaluation rules that apply to primitive operations. The execution of the operation is controlled by its definition and its execution properties.

Each statement in the definition is executed in sequence or as directed by branching statements. If the function has been defined with an explicit result, the last specification of the result parameter name is returned as the result of executing the operation. This result is then available for further evaluation of the expression in which the defined operation appears.

- Calling Sequence
- Suspension of Execution
- State Indicator
- Execution Properties

Calling Sequence

If a statement in a defined operation contains the name of a defined function or operator, that operation is called and flow of control passes to it. While the called operation is executing, the calling operation is said to be *pendent*, waiting to complete execution. If the called function or operator, in turn, calls another, it is pendent along with the original calling operation.

As each operation (or immediate execution expression) is invoked, it is placed in the *execution stack*. When execution of an operation or expression completes, it is removed from the execution stack.

The number of levels of called functions or operators is not limited except by the space available within the workspace. Pendent operations take up space in the execution stack; a sequence of called and calling operations may create a `WS FULL` condition if there is a large number of them or if any of them requires a sizable work area for calculation.

A function that calls itself is *recursive*. Local copies of the function behave as separate functions in the execution stack. Names localized by the function may have different values for each level on the stack.

Suspension of Execution

A defined operation becomes *pendent* if it calls another one. This is not the same thing as being *suspended*. Execution of a defined operation may be suspended in either of two ways:

- By an attention
- By an interrupt or an error

Using the **Signals** menu of the APL2 session manager, you can suspend execution of a defined operation (or an expression) by signaling:

Attention
to suspend execution at the end of the current statement being executed. (Unless an operation with the ignore attention property is active or pendent.)

Interrupt
to cause the system to behave as though an error were encountered.

If an error is encountered in a statement during execution of the defined operation, or if an interrupt is signaled, execution of the operation is suspended immediately and a message and the suspended operation are displayed. For example:

```
      ⎕CR 'F'
Z←F X
Z←10÷X
      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^  ^
```

As with other errors, ⎕EM and ⎕ET are set, and ⎕EA or ⎕EC can be used to recover from the error.

State Indicator

The *state indicator* is a term applied to a number of user views of the content of the execution stack. The views are presented in the calling sequence of defined operations, beginning with the most recently invoked, and may include:

- The names of the defined functions and operators
- The names which were localized when each operation was invoked
- The currently uncompleted statement for each operation level
- The line numbers within the operation definition for those statements
- Current scan and evaluation positions within those statements

Statements entered as input, rather than taken from defined operations, show an asterisk rather than operation name and line number.

If Editor 1 is in use, operations that are in definition mode may also be shown.

The ⎕LC system function provides limited information about the state indicator. More complete information is available using the system commands `)SI`, `)SIS` and `)SINL`:

```
      ⎕CR 'F'
Z←F X
Z←10÷X
      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^  ^
      )SI
F[1]
⋆
      )SIS
F[1] Z←10÷X
      ^  ^
⋆   F 0
   ^
      )SINL
F[1]    Z        X
⋆
```

While an operation is suspended, local names are available for inspection. However, any global values associated with those names are hidden or *shadowed*.

Statements remain in the state indicator until they have been cleared. If a workspace that has items in the state indicator is saved, the state indicator is also saved. There are several ways, listed below, to clear the state indicator. The one you use depends on what you are trying to accomplish and the situation that caused statements to be put in the state indicator.

- Escape
- `)SIC`
- Resume or Restart Execution
- Caution on Reinvoking Suspended Operations

No matter what technique is used to modify the state indicator, `⎕EM` and `⎕ET` are always set to values appropriate to the statement at the top of the state indicator.

Escape

Escape (→), a branch arrow with no expression to its right, abandons further attempts to execute the suspended function and the calling sequence that led to its being invoked. Escape clears the state indicator down to and including the next `⋆`. For example:

```
      )SI
F[1]
⋆
      →
      )SI
```

You can then correct the error and restart the function.

```
      ∇F[1] 'Z←''Cannot divide by 0''' ⎕EA 'Z←10÷X'∇
      F 0
Cannot divide by 0
```

Because only one calling sequence was in the state indicator, a single → cleared it. This is not the case in the following example:

```
      )SI
D[1]
⋆
B[2]
⋆
      →
      )SI
B[2]
⋆
      →
      )SI
```

)SIC

The system command `)SIC` clears the state indicator entirely. `)SIC n` clears `n` lines from the display of the state indicator.

```
      )SI
D[1]
⋆
B[2]
⋆
      )SIC 1
⋆
B[2]
⋆
      )SIC
      )SI
```

Note: `)RESET` is a synonym for `)SIC`.

Resume or Restart Execution

You may be able to modify variables or defined operations in the workspace and resume execution from the point at which it was halted by entering `→⍳0`.

```
      A←1
      N+2×A
VALUE ERROR
      N+2×A
      ^^
      A←0
      N←3
      →⍳0
5
```

Note that the new value of `A` was *not* used. APL2 had already computed `2×A` and was waiting to do the addition. The `→⍳0` resumed at that point, in mid statement.

Alternatively, for an error within a defined operation, you can correct the line in error by editing the operation, and then enter `→⎕LC` to direct the system to start over for the current line. Of course this is only one possibility; you could use any valid form of the branch expression.

```
      ⎕CR 'F'
Z←F X
Z←10÷X
      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^   ^
      ∇F[1] Z←'⌊/⍳0' ⎕EA '10÷X'∇
SI WARNING
      →1
1.797693135E308
```

The expression `⌊/⍳0` produces the largest representable number.

The message `SI WARNING` is displayed when editing affects a line of an operation appearing in the state indicator (if you edit the line or delete or insert lines before it).

Do Not Resume Execution by Invoking the Operation Again

If you correct the error in the operation and then invoke the operation again, the state indicator is *not* cleared. After the operation has executed, the earlier uncorrected version remains in the state indicator.

```
      ⎕CR 'F'
∇Z←F X
[1]Z←10÷X∇
      F 0
DOMAIN ERROR
F[1]  Z←10÷X
       ^  ^
      ∇F[1] Z←'⌊/⍳0' ⎕EA '10÷X'∇
SI WARNING
      F 0
1.797693135E308
      )SIS
F[1]  Z←'⌊/⍳0' ⎕EA '10÷X'
      ^ ^
⋆ F 0
  ^
```

If you decide not to resume the current operation, use `→` or `)SIC` to clear the state indicator before invoking the operation a second time.

Note: This also applies to expressions entered directly from the keyboard.

Execution Properties

A defined operation has four execution properties, which can be set *independently* with dyadic `⎕FX`, or all set on at the same time using ⍫ to open or close function definition mode. The following describes the execution effect of setting each property.

`1 0 0 0`	The defined function or operator may not be displayed or edited using the APL2 editors; may not be extracted using `⎕CR` (Character Representation) or `⎕TF` (Transfer Form); and may not be traced.
`0 1 0 0`	The defined function or operator is not suspended by an error or an interrupt. Instead the defined operation is terminated and the signal is raised in the environment from which it was called. See Suspension of Execution for additional information.
`0 0 1 0`	The defined function or operator ignores attentions and stop control settings during its execution.
`0 0 0 1`	The defined function or operator converts any error other than a resource error into a `DOMAIN ERROR`. (`INTERRUPT`, `SYSTEM ERROR`, `WS FULL`, and `SYSTEM LIMIT` are classified as resource errors.)

The execution properties of a called function or operator during an execution sequence are determined by "or-ing" its properties with those of the calling function or operator. For example, suppose function `F` has the nonsuspendable property (`0 1 0 0`) and function `G` has the error conversion property (`0 0 0 1`). If `F` calls `G`, both the nonsuspendable property and the error conversion property are imposed on `G` (`0 1 0 1`). Because execution properties are inherited by called functions and operators, if a locked function calls an unlocked function, the unlocked function behaves as though it were locked.

The execution properties of a defined operation can be inspected by using ⎕AT (Attributes). Execution properties can be changed only by using ⎕FX, and only if the operation can be displayed. Note that the definition of ⎕CR is such that, for displayable operations, the properties can be set as follows:

```
      properties ⎕FX ⎕CR 'program'
program
```

The default function or operator definition provided by the APL2 editors has *none* of these properties.

Debug Controls

APL2 includes two facilities for analyzing the behavior of defined functions and operators:

- Trace Control
- Stop Control

Trace Control

A *trace* is an automatic display of information generated by the execution of each selected line of a defined function or operator. If a trace request is in effect for a statement, the following information is displayed whenever the statement is executed:

- Function or operator name
- Line number in brackets
- Final array value (or branch) produced by that statement

Trace on a line containing multiple expressions separated by diamonds causes trace output for each expression evaluated.

The trace control for a defined operation is designated by prefixing `T∆` to its name. For example, a trace may be set on lines `1`, `3`, and `6` of a defined operation `RS` by entering:

```
      T∆RS←1 3 6
```

A trace may be set on all lines by entering:

```
      T∆RS←⍳n
```

where `n` is at least as great as the number of lines in the operation.

A trace is turned off by setting the trace control to `⍳0`.

Global names beginning with `T∆` may not be used for any purpose other than trace control.

Trace controls can be both set and referenced. A reference to a trace control vector returns only valid line numbers (in increasing order) upon which a trace has been set.

Trace settings are ignored if the "nondisplayable" execution property is set.

Stop Control

A defined operation can be made to stop during execution. If a stop request is in effect for a statement, processing stops just before the statement is to be executed, and the following information is displayed:

- Operation name
- Line number in brackets

Execution may be resumed by entering a branch statement.

The stop control for a defined operation is designated by prefixing `S∆` to its name. For example, a stop may be set on lines `1`, `3`, and `6` of a defined operation `RS` by entering:

```
      S∆RS←1 3 6
```

A stop may be set on all lines by entering:

```
      S∆RS←⍳n
```
where `n` is at least as great as the number of lines in the operation.

A stop is turned off by setting the stop control to `⍳0`.

Global names beginning with `S∆` may not be used for any purpose other than stop control.

Stop controls may be both set and referenced. A reference to a stop control vector returns only valid line numbers (in increasing order) upon which a stop has been set.

Stop control settings are ignored if the execution property "ignore weak attention" is set.

Error Reports and Error Codes

- Interrupts and Errors in APL2 Expressions
- Interpreter Messages
- Interpreter Event Codes

Interrupts and Errors in APL2 Expressions

Interrupts and errors in expressions typically result in three lines of output:

- A brief event description
- The expression that was interrupted or is in error
- Two carets pointing to tokens within the expression

Here is an example of an incorrect APL expression being entered, and the output that results:

```
      2÷'X'      Expression as entered
DOMAIN ERROR     Event description
      2÷'X'      Expression in error
      ^^         Carets pointing to the expression
```

The left caret indicates how far APL2 interpreted the statement from right to left. The right caret indicates where the error was detected or the interrupt occurred. In the above example, APL2 interpreted the entire expression. The error was detected by the Divide primitive because its right argument was not numeric.

Sometimes only one caret is seen because the point where the error was detected and the point to which APL2 had interpreted the expression are the same.

The error message can be retrieved using `⎕EM` (event message). Further information on the category of error can be obtained using `⎕ET` (event type).

When an error or interrupt occurs while a defined function or defined operator is running, APL2 displays an error message similar to that shown above for statements entered by the user. The difference is that the name of the operation and the line number precede the display of the statement in error.

Here is an example of an interrupt signaled by the user, because the function seemed to be taking too long. The reason for the problem was that the function was waiting for a value to be set in shared variable `CTL`.

```
      GETLOCK 'DBASE'    Function started
INTERRUPT                Event description
GETLOCK[7]  Z←CTL        Statement where event occurred
              ^          2 pointers on top of each other
```

Interpreter Messages

The following messages are listed alphabetically by event description. An information panel is available for each of them. See Interrupts and Errors in APL2 Expressions for a general discussion of interpreter message format and content.

- `⎕xx ERROR` (implicit errors)
- `AXIS ERROR`
- `CLEAR WS`
- `DEFN ERROR`
- `DOMAIN ERROR`
- `IMPROPER LIBRARY REFERENCE`
- `INCORRECT COMMAND`
- `INDEX ERROR`
- `INTERRUPT`
- `LENGTH ERROR`
- `LIBRARY I/O ERROR`
- `NOT COPIED`
- `NOT ERASED`
- `NOT FOUND`
- `NOT SAVED, THIS WS IS wsid`
- `NOT SAVED, LIBRARY FULL`
- `RANK ERROR`
- `SI WARNING`
- `SYNTAX ERROR`
- `SYSTEM ERROR`
- `SYSTEM LIMIT`
- `VALENCE ERROR`
- `VALUE ERROR`
- `WS FULL`
- `WS INVALID`
- `WS TOO LARGE`

Implicit Errors

This section covers a group of messages of the form

```
 ⎕xx ERROR
```

The first token in each message is the name of an APL2 system variable. It may be any one of the following:

`⎕CT ERROR`	`⎕ET ←→ 4 3`

A primitive function that uses `⎕CT` as an implicit argument was called when `⎕CT` had an inappropriate value or no value. `⎕CT` is an implicit argument of monadic `⌈` and `⌊`; and of dyadic `< ≤ = ≥ > ≠ ≡ | ∊ ⍳ ~` and `⍷`.

`⎕CT` must be a simple real scalar less than 1. It cannot be negative, but it can be zero, which indicates that all comparisons must be exact.

⎕FC ERROR	⎕ET ←→ 4 4

⎕FC had an inappropriate value or no value when dyadic Format (L⍕R) was called. Format by Example (character left argument) depends on the first five items of ⎕FC. Format by Specification (numeric left argument) depends on the first, fourth, and sixth items of ⎕FC.

⎕FC must be a simple character vector. Only its first six characters are significant. If shorter than six, defaults are used for the unspecified characters. The defaults and meanings of the six positions, in order, are:

[1]	.	Character for decimal point
[2]	,	Character for thousands indicator
[3]	⋆	Fill for blanks with pattern code 8
[4]	0	Fill for DOMAIN ERROR overflows
[5]	_	Pattern code character to be replaced by blank
[6]	¯	Indicator for a negative number

⎕IO ERROR	⎕ET ←→ 4 2

An attempt was made to execute a primitive function that uses ⎕IO as an implicit argument when ⎕IO had an inappropriate value or no value. ⎕IO is an implicit argument of bracket indexing; dyadic ⌷ ⊃ and ⍉; and the monadic and dyadic forms of ⍒ ⍋ ⍳ and ?.

The only acceptable values for ⎕IO are 0 and 1.

⎕PP ERROR	⎕ET ←→ 4 1

An attempt was made to use monadic Format (⍕R) or to display an array when ⎕PP had an inappropriate value or no value.

⎕PP must be a positive integer. Values larger than 16 are treated as if 16 had been specified.

⎕PR ERROR	⎕ET ←→ 4 7

An attempt was made to use the ⍞ system variable to create a character prompt immediately followed by a request for character input. However, ⎕PR has no value or an inappropriate one.

⎕PR must be either an empty character vector or a single character.

⎕RL ERROR	⎕ET ←→ 4 5

The Roll or Deal primitive (monadic or dyadic `?`) was called when `⎕RL` had an inappropriate value or no value.

`⎕RL` must be a positive integer less than `¯1+2⋆31`.

AXIS ERROR

`AXIS ERROR`	`⎕ET ↔ 5 6`

Bracket notation is being used to the right of a function or operator, and one of the following problems exists:

- The function or operator is not defined with an axis. This would include any defined function or operator. It also includes primitive operators except for `/` or `\`; and non-scalar primitive functions unless a *with Axis* form is explicitly defined for them. (See Scalar Functions and the Alphabetic List of Functions.)
- The indicated axis is incompatible with the function or operator and the given arguments. This includes cases where the axis is not a valid integer (except Catenate, which allows decimal values), is not an element of `⍳⍴⍴array`, or produces derived function arguments which are not compatible.
- The axis specification includes semicolons. Semicolons can be used inside brackets only when the notation is used for indexing an array. The token to the left of the bracket notation is currently a function or operator. It may be that it was intended to be an array.

CLEAR WS

```
CLEAR WS
```

The current active workspace was replaced with a clear workspace. This is an informational message, not an error.

DEFN ERROR

```
DEFN ERROR
```

The `∇` or an editing command was misused:

- A syntactically incorrect `∇` or `⍫` command was entered to begin edit mode.
- An invalid character was used outside of a quoted string or comment.
- The object cannot be edited. For example, a numeric variable or a locked function.
- An invalid edit command was entered (inside brackets) while in function edit mode.
- The closing `∇` or `⍫` was entered to establish an invalid object.
- A `∇` or `⍫` was entered on an unnumbered line to close a definition.
- An attempt was made to name an object with a name already in use in the active workspace.

DOMAIN ERROR

`DOMAIN ERROR`	`⎕ET ←→ 5 4`

This error is signaled in any of the following cases:

- The data type of an argument or operand is invalid for the primitive operation it is being given to. This includes character data when only numeric data is allowed. It also includes unsupported types of numeric values, such as decimal values when integers are required, values other than `0` or `1` for Boolean operations, etc.
- The depth of an argument or operand is invalid for the primitive operation it is being given to. The Depth primitive (monadic ≡) can be used to check the depth, or the `DISPLAY` function from the workspace of the same name can be used to get a visual representation of the data.
- A calculation requires or produces data that is beyond the range of the system implementation but does not fit any of the categories of `SYSTEM LIMIT` (this can occur with some mathematical functions).
- A nonresource error occurred in a defined function or operator whose fourth execution property is set to convert nonresource errors to a `DOMAIN ERROR`. (See dyadic `⎕FX`).
- A derived function from the slash operator or inner product was presented with an empty argument but no identity function existed for the function operand.

IMPROPER LIBRARY REFERENCE

`IMPROPER LIBRARY REFERENCE`

An undefined library number was specified for any of the system commands which accept library numbers. Library numbers are defined using environment variables or `apl2.ini` keyword definitions of the form `APL`*`nnnnn`*. The notation *`nnnnn`* shown here must always be replaced by exactly 5 digits. Here is an example of a definition for library 123:

```
APL00123=C:\MYSTUFF\LIB123
```

INCORRECT COMMAND

`INCORRECT COMMAND`

The APL2 system command entered is invalid or has invalid parameters. See System Commands for the names and syntax of the supported commands.

INDEX ERROR

INDEX ERROR	⎕ET ↔ 5 5

An index specified for Bracket Indexing (`R[I]`), Index (`I⌷R`), or Pick (`I⊃R`) is outside the bounds of the array (`R` as shown here). Depending on `⎕IO`, those bounds are either `1` to `n` or `0` to `n-1`, where `n` is the length of the array along the dimension being indexed.

INTERRUPT

INTERRUPT	⎕ET ↔ 1 1

An interrupt was signaled during processing, and execution is halted. Execution can be resumed with `→⍳0` or restarted by branching to a line number in the defined operation. If execution is not resumed or restarted, the state indicator should be cleared using `→` or `)SIC`.

LENGTH ERROR

LENGTH ERROR	⎕ET ↔ 5 3

Two arguments to a function (or perhaps an argument and an operand of an operator) have lengths which are incompatible with respect to an axis along which the values are being processed. A length of 1 is compatible with any length by scalar extension.

Note that for the Expand operator (`LO\R`), the requirement is that the number of `1`'s in the operand must be compatible with the length of the right argument axis along which it is being applied. Scalar extension applies only if the operand is of length 1.

LIBRARY I/O ERROR

LIBRARY I/O ERROR

An access error is preventing successful completion of a system command that is attempting to read or write a workspace or transfer file. The message usually arises while writing, and may indicate either a write-protected device or an undefined directory path.

NOT COPIED

NOT COPIED: object-names

The listed objects were not copied for one of the following reasons:

- A `)PCOPY` was issued, and the objects already exist in the active workspace.
- The objects do not fit in the active workspace.
- `)IN` finds objects which are not valid transfer forms. One possible reason is that an APL2 mainframe transfer file contains underbarred letters, or was downloaded with EBCDIC/ASCII conversion.
- `)OUT` could not write explicitly named objects because they do not exist in the active workspace.
- `)OUT` could not write explicitly named objects because they are either system names or locked user programs. (See the "no display" attribute of `⎕FX`.)

NOT ERASED

```
NOT ERASED: object-names
```

The listed objects were not erased by the `)ERASE` command because the objects do not exist in the active workspace.

NOT FOUND

```
NOT FOUND
```

One of the following problems was found:

- The workspace specified with a `)COPY`, `)DROP` or `)LOAD` command does not exist.
- The file specified by the `)IN` command cannot be found or is not a transfer file.

NOT SAVED, THIS WS IS wsid

```
NOT SAVED, THIS WS IS wsid
```

One of the following problems was found:

- The `)SAVE` system command was issued in a `CLEAR WS` with no specified workspace name. Reissue the `)SAVE` command, and include the name under which you want the workspace to be saved.
- The workspace named in the `)SAVE` command exists in the library but is not the same as the name of the active workspace. If you want to replace the saved workspace, issue the `)WSID` command first, changing the name of the active workspace to match.

NOT SAVED, LIBRARY FULL

```
NOT SAVED, LIBRARY FULL
```

The disk to which you are attempting to save the workspace or transfer file is full, or so nearly full that there is not enough room for the workspace. You can identify the disk involved as follows:

- If the `)SAVE` or `)OUT` command provided a path and file name in quotes, or if `)SAVE` provided no name and `)WSID` displays a quoted name, then the drive is either the one explicitly listed at the beginning of the path or the drive from which APL2 was started.
- If the `)SAVE` or `)OUT` command provided an unquoted workspace name, or if `)SAVE` provided no name and `)WSID` displays an unquoted name, then the drive is defined by a library setting for `APLnnnnn` where `nnnnn` is the library number provided on the command (extended to 5 digits) or `01001` if no library number is given. The library definition either lists the drive explicitly, or it is the drive from which APL2 was started.

You may need to take one or more of the following actions to recover:

- Erase unneeded files on the drive that is full.
- Save the workspace on a different drive. See the comments above on how the drive is determined.
- If an older version of the workspace or transfer file already exists, `)DROP` it first, and then reissue the `)SAVE` or `)OUT`.

  Warning: Remember that it is possible to lose your workspace altogether if you do this. There might be a power failure before you have a chance to save the new copy, or it might be that the new version is enough larger that it won't fit even after dropping the old one.

- Try saving the workspace in a different format. In many cases transfer files created by `)OUT` take less space than workspaces stored by `)SAVE`. In some cases the reverse is true.
- Don't forget that for reasonably small workspaces you could also save to a removable diskette. Even very large workspaces can often be saved to diskettes by using `)OUT` and naming objects explicitly.

RANK ERROR

`RANK ERROR`	`⎕ET ↔ 5 2`

An array specified as an argument or operand has a rank that is incompatible with another argument or operand. If the array is nested, the incompatibility may exist below the top level of structure.

SI WARNING

```
SI WARNING
```

A defined function or operator was altered while it was on the execution stack. There are two basic ways that programs can be altered:

- If the function or operator was replaced by the `)IN`, `)COPY` or `)PCOPY` command; or by the `⎕FX` or `⎕TF` system function, the copy used for future calls is changed, but the copy already being executed is left as it was. No `SI WARNING` is needed or produced.

  Note: The `EDIT` workspace uses `⎕FX`, so it behaves this way. This is also true at present if the object was edited using ∇-edit when `)EDITOR xxx` is in effect, for example `)EDITOR APLEDIT`, but that behavior may change in the future.

- If the function or operator was modified using Editor 1 or the session manager's object editor, the copy already being executed is modified as well as the permanent copy. `SI WARNING` may or may not be produced, depending on the nature of the changes. If the changes don't affect resumption of the statement currently being executed they may be accepted without displaying the message.

When a program has been suspended there are two ways to get it going again. You may either *resume* it or *restart* it.

resume
: Begin executing exactly where the program was stopped. This is often in the middle of a line, and can be done by entering `→⍳0`.

restart
: Begin execution with some complete statement of the current function. This is done by entering `→⎕LC` to restart at the beginning of the line that was being executed.

If changes are made to the copy on the stack, and those changes make it impossible to resume the statement currently being executed, then any attempt later to do so will also result in `SI WARNING` at that point in time. This can occur if:

- The function is suspended (stopped while it was executing) in mid-statement, and `→⍳0` is used to resume it.
- The function is pendent (has called another function which has not returned to it) and the function it called then does return.
- The function is either suspended or pendent, and the update makes local names become global, or global names become local. This may make it impossible to resume or restart the program at all.

SYNTAX ERROR

`SYNTAX ERROR`	`⎕ET ←→ 2 n`

The displayed APL2 expression is constructed improperly. The second item of `⎕ET` provides information about the type of problem diagnosed:

`⎕ET`	**Description**
`2 1`	Required operand or right argument omitted

`2 2`	Ill-formed line, for example unmatched parentheses or brackets
`2 3`	Invalid name class, for example assigning a value to a constant or label
`2 4`	Invalid operation in context

SYSTEM ERROR

`SYSTEM ERROR`	`⎕ET ←→ 1 2`

A fault occurred in the internal operation of the APL2 system, or the active workspace was damaged. This can be caused by a workspace file being partially overwritten, or by an improperly coded auxiliary processor or Processor 11 external routine. But if none of those conditions are suspect, you should report the problem to IBM as a possible error in the APL2 product.

SYSTEM LIMIT

`SYSTEM LIMIT`	`⎕ET ←→ 1 n`

The requested operation or action exceeds some implementation limit. See the implementation limits documented in the *APL2 User's Guide*. Note that if the limit exceeded involves shared variables, you may be able to avoid the restriction using parameters in the `apl2svp.prm` file.

VALENCE ERROR

`VALENCE ERROR`	`⎕ET ←→ 5 1`

An attempt has been made to specify a left argument for a monadic function, or to specify a single argument for a dyadic function.

Note: Defined functions whose header shows a left argument are actually ambi-valent, rather than dyadic. This means that the left argument is optional. Functions can be coded to allow for an omitted left argument by checking its name class. For example:

```
   →(0=⎕NC leftarg)/NOLEFT   ⍝ Go handle monadic call
```

If this check is not made, an omitted left argument will typically result in a `VALUE ERROR` during execution rather than a `VALENCE ERROR` during function call. To avoid this, functions which must be called dyadically should include a statement like this:

```
   ⎕ES (0=⎕NC leftarg)/5 1  ⍝ VALENCE ERROR if no left arg
```

VALUE ERROR

`VALUE ERROR`	`⎕ET ←→ 3 n`

There are two cases which you can distinguish by the value of `⎕ET`:

`⎕ET`	**Description**
`3 1`	The variable being referenced does not currently have a value.
`3 2`	An attempt was made to use the result of a function that does not return a result.

Note: `VALUE ERROR` may also occur while executing a defined function that expected two arguments but was called with only a right argument. See VALENCE ERROR for suggestions on this.

WS FULL

`WS FULL`	`⎕ET ←→ 1 3`

An operation requires more workspace storage than is currently available. The workspace size is controlled by the `-ws` invocation option, which defaults to 10 megabytes. Note that the workspace size limit is independent of the amount of real storage on the computer, or even of the amount of available disk space you have.

If the workspace size exceeds the amount of *available* real storage, the operating system will swap parts of it, or other things in real storage, to disk. This can cause dramatic losses in performance, but does not cause `WS FULL` and is not under APL2's control.

If the amount that needs to be swapped exceeds the *available* space on the drive containing the swap file, the system will go into emergency recovery mode, and may crash. This will still not cause `WS FULL` (though it is likely to be much worse) and is not under APL2's control.

It is your responsibility to choose a workspace size which is large enough to avoid `WS FULL` but not so large that it exceeds swap capabilities. For good performance you should choose a size significantly smaller than the amount of real memory you have. Practical limits depend on how you have customized your system, and what applications are sharing it with APL2.

Sometimes a trivial change in an APL algorithm can cause a massive change in storage requirements. For example, a Boolean array can be stored eight elements per byte. Including a single non-Boolean value in the array (a `2` or `¯1`, for example) causes the entire array to take 32 times as much space as it did before. Intermediate results may also require much more storage than the values ultimately saved. An outer product, for example, may use a great deal of storage, even if it is immediately reduced to a much more manageable array.

WS INVALID

`WS INVALID`

The `)LOAD` or `)COPY` command was issued to access a file that is not a valid APL2 workspace. Possible causes include:

- transfer from another system through an intermediary that performed ASCII or line-end conversion.
- a workspace that was saved by APL2 on a different operating system than you are currently running. This includes workspaces written by APL2/PC on the same system, as well as workspaces written on other types of systems by IBM APL2 products. Such workspaces can only be processed if they are read as transfer files, i.e. written using `)OUT` and read using `)IN`.
- a file written or modified by something other than the APL2 interpreter.

WS TOO LARGE

```
WS TOO LARGE
```

The `)LOAD` or `)COPY` command was issued to access a file that is larger than the size currently allocated for the APL2 workspace.

In order to access the file APL2 must be invoked with a workspace size great enough to contain the file. The `-ws` invocation option can be used to set the workspace size.

Interpreter Event Codes

The following system functions return codes which indicate whether they were successful, or what kinds of problems they encountered:

`⎕EC`	Execute Controlled
`⎕EX`	Expunge
`⎕FX`	Fix
`⎕NA`	Name Association
`⎕NC`	Name Class
`⎕SVO`	Shared Variable Offer (Inquire)

See also `⎕ET` - Event Type for a system variable containing event codes.

Glossary

argument
: An argument is data that is being passed to a function. It may be a single item, a list of items, or an n-dimensional array. Each item may be a number, a character, or a collection of subitems, and these kinds of items may be mixed in an array.

dyadic
: A function which accepts both a left and right argument, or an operator which requires both a left and right operand.

function
: A function accepts argument data and transforms it in some way to produce result data. It is possible in APL for user functions to be niladic, i.e. to be passed no data explicitly, but most functions are either monadic, requiring one array as a right argument; or ambi-valent, requiring a right argument array and permitting a left argument array.

monadic
: A function which accepts only a right argument, or an operator which accepts only a left operand.

operand
: An Operand is a function, or occasionally data, that is being passed to an operator. The function may be a primitive, system, or defined function.

operator
: An operator accepts functions or occasionally data as operands and applies its operands in some way to produce a derived function. For example, the Reduction operator can be given the Plus function as an operand, to produce a derived function that will perform a summation over the items within its argument data. An operator must be defined as either monadic, requiring one function or array as a left operand; or dyadic, requiring both a left operand and a right operand. Either way, the derived functions they produce can be monadic or ambi-valent, like other functions.

primitive
: An APL term referring to functions and operators which are built in to the language, and are represented by a special one-character symbol.

progression of integers
: Most APL systems (including this one) provide an optimized format for storing arrays that are integral progressions initially created using ⍳. These would include:

```
A←⍳1000000
B←5×⍳2000000
C←¯8+⍳3000000
D←¯4×9+⍳4000000
```

Altogether there are ten million items in those four arrays, yet they take about as much space as a dozen or so fractional numbers in a single array! You never *have* to worry about things like this to make your APL functions work correctly, but it's something to keep in mind when working with large arrays.

shared variable
: The APL2 system provides facilities which makes it possible for two independent processes to cooperate by sharing common data. ⎕SVC provides controls so that the processes can serialize their access to the data using any of a number of protocols. The processes can be an APL session and an Auxiliary Processor, or two APL sessions. Using APL2's cooperative processing facilities, the processes may be running on different machines.

state indicator
: During processing, APL2 must, of course, keep track of function calling sequences so that when one function completes it can resume execution of its caller. The information required for this includes an

executable image of the function, the position within the function from which the call was made, and the current values of any local variables. The accumulation of this information for all active calling levels is the state of the system, and various parts of it can be queried using the `⎕LC` function and the `)SI`, `)SINL`, and `)SIS` system commands. The state can be partially or fully reset using → or the `)SIC` system command.

system command

System commands are intended for communication between the user and the APL2 system. They cannot be issued directly by an APL program, and use a syntax beginning with a right parenthesis which is intentionally chosen to be invalid as an APL statement.