APL Special Edition Documentation

The documentation in this workspace consists of six chapters and a reminder of how to print. The name of the APL variable, which is a character vector, is: To_Print

If you want to read this reminder, simply type

```
      To_Print
```

at the prompt. APL names are case sensitive, so you must type it exactly as above.

The documentation chapters include:

SE_CH01 Terminology and the Keyboard
SE_CH02 Using APL
SE_CH03 Writing Your Own Functions
SE_CH04 Function Reference -- Arithmetic Functions
SE_CH05 Function Reference -- Structural Functions
SE_CH06 Function Reference -- General Functions, Operators and Other Symbols

If you print out just Chapters 1 and 2, you can get started using APL.
→

APL Special Edition Documentation

Reminder of How to Print

To print a chapter of the documentation, or indeed any variable in a workspace, you must:

At the DOS prompt,
    run APLPRINT
    Start APLSE

In APL,
)load PRINTERS
    run SELECT   ⍝ function names are case sensitive
        (Within SELECT, you select a printer type and port)
)load INITIAL
    type 10 ⎕arbin varname

       where varname is the name of the variable you want to print, for example
       SE_CH06 -- You get the quad symbol (⎕) by typing Alt+L.

     With some laser printers, you may want to catenate a Form Feed character
     to force the last page, so your statement looks like this:

      10 ⎕arbin SE_CH06,⎕tcff
→

APL Special Edition Documentation

Chapter 1. Terminology and the Keyboard

1.1 First, Some Terminology

When you start APL, you are in the session manager in immediate execution mode. Immediate execution mode means you can type a single statement and have the system evaluate (execute) it. For example, type: 3+5 (and press Enter).

Note that you start at the six-space indent; the system returns its output, if any (you can think of it as an answer, a response, or a result), at the left margin.

The alternative to being in the session manager is to be in an edit session, where you can create a program or a variable. You can create a program of your own (called a function in APL) containing multiple statements, which the system will run (execute) as a unit. Or you can create a named variable, which can be of various sizes, shapes and content. (More about this later.)

The session manager is the interface between you, the user, and the system (the interpreter). The "container" for your work is called a workspace. You can name and save workspaces and reload them from disk. For example, the INITIAL workspace contains this documentation.

You manage workspaces with system commands, which start with a right parenthesis. When you loaded the INITIAL workspace, you typed:)LOAD initial

To remove the contents of a workspace from memory, type:)CLEAR
Do this now. This leaves the contents of INITIAL unchanged on your disk and provides an "empty" container for your subsequent work. After you create something in a workspace, if you want to save your work, type:)SAVE followed by the name under which you want to save the workspace file on your disk.

1.2 Finding, Learning, and Using the APL Font

Before you started this session, you ran the command aplfont. APL uses a number of special symbols to manipulate data. Some of these symbols are part of the normal keyboard. For example, you use the asterisk (star) to raise a number to a power. The statement 2*4 returns 16. In this example, the function represented by the asterisk has a left argument and a right argument (2 and 4 respectively). This is called a dyadic function.

When you use the asterisk with only a right argument, the result is the value e raised to the power of the right argument. The statement *2 gives e-squared. In this case, the asterisk is a monadic function. Thus, you can think of the asterisk as two different functions: Used monadically, the result is exponentiation; used dyadically, the result is a power. (You can also think of it as one function with a default left argument.)

Many of the APL symbols have multiple uses. Those symbols that are not part of the normal keyboard are part of the aplfont you installed to start this session. You can generate those symbols by pressing the Alt key plus another key from the keyboard or by pressing Shift+Alt+ another key. There is a table at the end of this chapter that shows the keys and symbols. Chapters 4-6 provide a summary of each of the primitive functions.

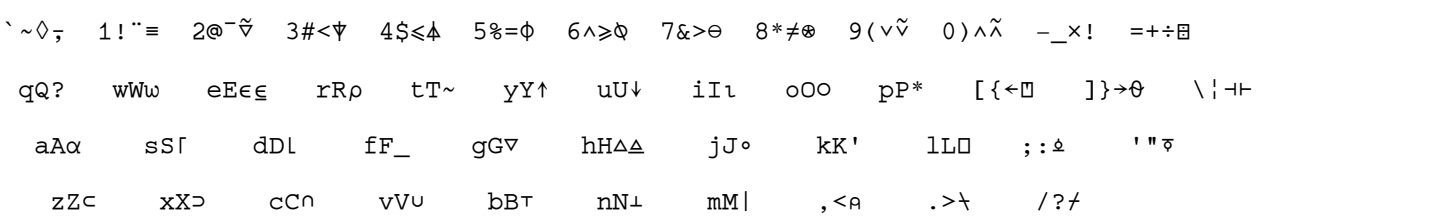
1.3 Moving Around in the Session

The session is the location where you type APL statements in immediate execution mode. It is also a record of what you have done and the results you obtained. You can scroll through the session with the cursor keys. There are many other keys and keystroke combinations that you can use. If you type Ctrl+H, there is a series of help panels that summarize the various keystrokes.

One feature of particular note: You can edit any line in the session and execute it simply by pressing the Enter key with the cursor on that line. You do not have to retype the whole line or move it to the bottom of the session. This allows you to correct mistakes easily or experiment with statements.

1.4 The Keyboard

The diagram below is a rough approximation of the location of the APL symbols on a standard 101-key layout keyboard. In a group of three symbols, you get the not-so-obvious one by pressing Alt+ the key. If there is a second not-so-obvious symbol, press Shift+Alt+ the key. The list that follows the diagram shows the various APL symbols along with the name used to refer to each symbol and the names of the primitive functions it represents or the use for the symbol. If there are two functions, the first one is usually the monadic function. The list is arranged in order of the normal keyboard from top to bottom, left to right, but separating the letter keys and symbol keys.



---------- TOP ROW KEYS ----------

UNSHIFTED	SHIFTED	ALT+	SHIFT+ALT+
` {left of 1} {not used}	~ tilde (logical) NOT Without	◊ diamond {statement separator}	⍪ catbar Catenate {first}
1	! bang Factorial Binomial	¨ {not used in APLSE}	≡ equivalent Match
2	@ at sign {not used}	¯ high minus {designates a negative number}	⍫ del-tilde Lock function {use with caution!}
3	# octothorpe {not used}	< less than Less Than	⍱ downgrade Numeric Grade Down Character Grade Down
4	$ dollar sign {not used}	≤ less than/equal Less Than or Equal	⍋ upgrade Numeric Grade Up Character Grade Up
5	% percent {not used}	= equal Equal	⍉ rotate Reverse {last} Rotate {last}
6	^ caret (logical) AND	≥ grtr than/equal Greater Than or Equal	⍒ transpose Transpose
7	& ampersand {not used}	> greater than Greater Than	⊖ rotate bar Reverse {first} Rotate {first}
8	* asterisk Exponential Power	≠ not equal Not Equal	⍟ logarithm Natural Logarithm Logarithm
9	(left paren {used to group	∨ or (logical) OR	⍲ nor (logical) NOR

or separate}

0	) right paren Recall statement {first char of system commands}	∧ and (logical) AND {same as Shift+6}	⍲ nand (logical) NAND
- minus sign Negate Subtract	_ underscore {used in names of objects}	× times Signum Multiply	! bang {same as Shift+1}
= equal Equal	+ plus sign Conjugate Add	÷ division sign Reciprocal Divide	⌹ domino Matrix Inverse Matrix Divide

---------- SECOND ROW SYMBOL KEYS ----------

UNSHIFTED	SHIFTED	ALT+	SHIFT+ALT+
[left bracket {used to index}	{ left brace {not used}	← left arrow Assign	⍞ Quote-quad Prompt {for character input/output to terminal}
] right bracket {used to index}	} right brace {not used}	→ right arrow Branch	⍬ zilde {empty vector}

[] Index Into
[]← Index Assign

\ backslash Expand {last} Scan (Operator)	¦ split stile Magnitude Residue	⊣ left tack {not used}	⊢ right tack {not used}

---------- THIRD ROW SYMBOL KEYS ----------

UNSHIFTED	SHIFTED	ALT+
; semicolon dimension separator	: colon {last character of label}	⍏ hydrant Execute
' single quote Character string delimiter	" double quote {not used}	⍕ thorn Format / Pattern Format

---------- BOTTOM ROW SYMBOL KEYS ----------

UNSHIFTED	SHIFTED	ALT+
, comma Ravel / Catenate {last}	< less than {same as Alt+3}	⍝ lamp comment designator
. period Inner Product operator {used with jot for ∘. Outer Product operator}	> greater than {same as Alt+7}	⍀ backslash-bar Expand {first} Scan {first} operator
/ slash Compress {last} Reduction {last} operator	? question mark Roll / Deal {same as Alt+Q}	⌿ slash-bar Compress {first} Reduction {first} operator

---------- SECOND ROW LETTER KEYS ----------

KEY	SYMBOL / NAME	FUNCTION(S)
Alt+Q	? question mark	Roll / Deal
Alt+W	⍵ omega	{not used}
Alt+E	∈ epsilon	Member Of
Shift+Alt+E	⍷ epsilon-underscore	Find

Alt+R	⍴ rho	Shape / Reshape
Alt+T	~ tilde	NOT / Without
Alt+Y	↑ up arrow	Take
Alt+U	↓ down arrow	Drop
Alt+I	⍳ iota	Index Generate / Index Of
Alt+O	○ circle	Pi Times / Trigonometric functions
Alt+P	* asterisk	Exponential / Power {same as Shift+8}

---------- THIRD ROW LETTER KEYS ----------

KEY	SYMBOL / NAME	FUNCTION(S) / USES
Alt+A	⍺ alpha	{not used}
Alt+S	⌈ ceiling	Ceiling / Maximum
Alt+D	⌊ floor	Floor / Minimum
Alt+F	_ underscore	{same as Shift+minus sign}
Alt+G	∇ del	Function Definition / {also used with lamp for public comments ⍝∇}
Alt+H	∆ delta	{used in names of objects}
Shift+Alt+H	⍙ delta-underscore	{used in names of objects}
Alt+J	∘ jot	(with period ∘.} Outer Product operator
Alt+K	' single quote	delimiter {same as unshifted quote key}
Alt+L	⎕ quad	Prompt for numeric (evaluated) input / Output to terminal / {also used as first character of system functions, variables, and constants}

---------- BOTTOM ROW LETTER KEYS ----------

KEY	SYMBOL / NAME	FUNCTION(S)
Alt+Z	⊂ enclose	{not used in APLSE}
Alt+X	⊃ disclose	{not used in APLSE}
Alt+C	∩ intersection	{not used}
Alt+V	∪ union	{not used}
Alt+B	⊤ decode	Base Value
Alt+N	⊥ encode	Representation
Alt+M	| stile	Magnitude / Residue

→

Chapter 2. Using APL

2.1 APL is Powerful

APL is a powerful programming language. One of its greatest strengths is that it handles entire arrays of data as single objects. This document explains some of the terms and functions of APL. It will also demonstrate many more without explicit definition in the course of explaining the basics. This document can get you started using APL and show you that there are many tools for you to use. The possibilities for learning are endless.

To get started, we will do some simple addition. You can execute all statements that are indented six spaces.

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

A string of numbers is called a vector. You can assign a single value or a string to a variable name. You use the left arrow for assignment.

```
      vec_a ← 11 9 7
```

You can add a scalar (one number) to a vector or you can add another vector of the same length. When you add a scalar, the scalar is added to each of the elements of the vector. When you add two vectors, the corresponding elements are added. Note that spaces are not necessary between a number and a function symbol; you must have at least one to separate numbers in a vector.

```
      vec_a+4
15 13 11
      vec_a +8 22.5 ¯3
19 31.5 4
```

Note that negative numbers are designated with a high minus and not the normal minus sign, which is used for subtraction. It is typical of APL that expressions have a distinct meaning. Thus, 5 ¯3 is a two-element vector, consisting of the values five and negative three; 5 -3 is an expression using the Minus function: its value is two. You cannot have a space between a high minus and its corresponding numeral.

Besides the many functions that perform arithmetic calculations on arrays, APL has structural functions that reorganize the data or select subsets of it.

```
      ⌽vec_a
7 9 11
```

Although APL expressions are unambiguous, many symbols do double duty. The phi symbol used monadically, as above, is called Reverse. If you use it dyadically, that is, with a left argument, it is called Rotate. Naturally, Rotate does something different than Reverse.

```
      1⌽vec_a
9 7 11
      2 ⌽ vec_a
7 11 9
      3 ⌽vec_a
11 9 7
```

Note that vec_a is unchanged by these manipulations. The result of applying a function to vec_a gives the result. You can assign a new value to vec_a with the assignment arrow in the same statement that manipulates it.

```
      vec_a
11 9 7
```

```
      vec_a ← vec_a,5 3 1
```

Note that there is no explicit result displayed when you assign the result.

```
      vec_a
11 9 7 5 3 1
```

You can create variables of greater dimensions. Reshape, the rho symbol used dyadically, allows you to specify how you want your data. To make a matrix, which is a two-dimensional array, use a vector of length two as the left argument to Reshape.

```
      mat_a ← 2 3 ⍴ vec_a
      mat_a
11 9 7
 5 3 1
```

The last dimension is the number of columns in the matrix. If you have only one dimension, that is, a vector, the dimension is the number of elements. The vector vec_a has six elements; the matrix mat_a has two rows and three columns.

Another often used primitive function is the index generator, monadic iota. It gives a sequence of numbers from one up to the value of the right argument.

```
      ⍳6
1 2 3 4 5 6
```

You can use iota to construct almost any regular sequence of numbers in an arithmetic or geometric progression.

```
      11-⍳10
10 9 8 7 6 5 4 3 2 1

      2*⍳4
2 4 8 16
```

As you see from the above statements, you can combine primitive functions in one statement.

```
      mat_b←2 3⍴⍳6
      mat_b
1 2 3
4 5 6
```

Try some simple arithmetic statements:

```
      mat_a + mat_b
      mat_a - mat_b
      mat_a × mat_b
      mat_a ÷ mat_b
      mat_a * mat_b
      mat_a ⌊ mat_b
```

Try some simple structural statements. (The lamp symbol (⍝) defines everything to the right as a comment. You will find this useful in writing functions.)

```
      mat_a , mat_b  ⍝ This new matrix has 2 rows and 6 columns
      mat_a ⍪ mat_b  ⍝ This new matrix has 4 rows and 3 columns
      ⌽mat_a
      ⊖mat_b
      ⍉mat_a         ⍝ This new matrix has 3 rows and 2 columns
```

Note that the first Catenate function (comma) works along the last dimension, the columns. The second Catenate function (catbar) works along the first dimension, the rows. This correspondence of functions is true for several other functions as well, for example, the Reverse functions (phi and theta).

You can also create arrays of greater dimension by catenating and adding a

dimension. You do this by specifying a fractional dimension with the function.

```
      mat_a,[0.5]mat_b
11 9 7
 5 3 1

 1 2 3
 4 5 6
```

This is a three-dimensional array with two planes, each of which has two rows and three columns. You could get the same array by specifying:

```
      2 2 3 ⍴ mat_a⍪mat_b
```

You can see the difference between this three dimensional array and the two-dimensional array for four rows and three columns that you created using just the catbar (⍪) by the extra blank row between the planes.

You can get other arrangements of the same data by inserting the new dimension at a different spot. Experiment with mat_a,[1.5]mat_b and mat_a,[2.5]mat_b.

You can use monadic rho (Shape) to determine or confirm the shape of results or variables. For example; ⍴mat_a,[1.5]mat_b

2.2 The Order of the Universe

The fundamental concept of understanding APL statements is that the order of execution is from right to left. In the last example, the comma is one function, Catenate, which takes a right argument and a left argument. The rho symbol, Shape, is another function, which takes (in this case) only a right argument. The Catenate is performed first; the result of that function becomes the argument to Shape.

Similarly, arithmetic functions are executed from right to left regardless of what they are. There is no hierarchy of functions.

```
      6+5×4
26
      5×4+6
50
      5×4,6
20 30
      6,5×4
6 20
      ⍴⍳6
6
```

Subtraction often surprises new users of APL; (6 minus 2 is evaluated first).

```
     9-6-2
5
```

You can use parentheses to change the order of execution.

```
      (9-6)-2
1
```

2.3 Operators

Operators are a special category of APL primitives that modify functions. Operators take one or more functions as an operand and the result is a derived function that acts on one or two arrays.

The Reduction operator reduces the rank of its array argument. For example, you can perform an arithmetic operation on a vector to get a scalar result. The symbols for Reduction are the slash (/) and the slash-bar (⌿). The first example below sums the values in the vector. The next examples multiply the values in a matrix either by row or by column.

```
      vec_a
11 9 7 5 3 1
      +/vec_a
36
```

```
      mat_a
11 9 7
 5 3 1
      ×/mat_a
693 15
      ×/⍉mat_a
55 27 7
```

Notice that the slash works along the last dimension. You can use slash-bar to work along the first dimension. The equivalent of the last example is:

```
      ×⌿mat_a
55 27 7
```

The Scan operator, whose symbols are the backslash (\) and backslash-bar (⍀), performs its function on successively greater portions of the argument array.

```
      ÷\⍳6
1 0.5 1.5 0.375 1.875 0.3125
```

An important key to understanding this operator is that the first element of the result is the first element of the argument (by definition); the second element of the result is the result of applying the function to the first two elements of the argument; and so on. However, for each calculation, the expression is evaluated from right to left. So, the second element of the result is 1÷2. However, the third element of the result is (2÷3) divided into 1. The last element of the result is:

```
      1÷(2÷(3÷(4÷(5÷6))))
```

As with other functions and the Reduction operator, the basic form operates along the last dimension, while the backslash-bar operates along the first.

```
      mat_b
1 2 3
4 5 6
      -⌿mat_b
 1  2  3
¯3 ¯3 ¯3
```

Subtraction is particularly tricky operating from right to left.

```
      -\mat_b
1 ¯1 2
4 ¯1 5
```

The other two operators are inner product and outer product. See Chapter 6 for a brief description of their uses, and then try experimenting with all the functions and operators.

2.4 Boolean functions and Selecting Data Items

Boolean functions are essentially tests of two items to see if they meet some criterion. If the answer is true, the result is 1; if not, the result is zero. The first example below compares two matrices, position by position to see if the value in the first is greater than the corresponding value in the second. The next example checks whether the value of the left argument, in this case 5, is contained in the right argument. The last example makes the same check position by position.

```
     mat_a > mat_b
1 1 1
1 0 0

     5∊vec_a
1
     5⍷vec_a
0 0 0 1 0 0
```

One common use for Boolean functions is to select a subset of data. To

demonstrate, we will create a matrix by multiplying our two sample matrices, ravel the result into a vector, and select the elements evenly divisible by 3.

```
      vec_c←,mat_a × mat_b
11 18 21 20 15  6
      Bool← 0 = 3 | vec_c
      Bool
0 1 1 0 1 1
      newvec←Bool/vec_c
18 21 15 6
```

And you can sort the resulting values in ascending order:

```
      finalvec←newvec[⍋newvec]
      finalvec
6 15 18 21
```

The statement using brackets is called indexing because it uses an index to identify parts of an array. You can select one or more elements of any array by specifying their indices.

```
      vec_a[2 4]
9 5
      mat_b[2; 3]
6
```

Note that the index to a matrix requires two values to specify one element, in this case the second row, third column.

2.5 Character Data

While APL is extremely powerful manipulating numeric arrays, it is likewise powerful manipulating character data. Although you cannot add letters, the structural functions and Boolean functions enable you to do dramatic things with very few statements.

You specify a character variable by putting single quotes around your data.

```
      char_vec_a←'How do you do?'
      ⍴char_vec_a
14
```

How many times does the letter o appear in your vector?

```
      +/'o'=char_vec_a
4
```

The last example, although compact, uses a Boolean function and an operator. If you have trouble understanding how this works, break it down into steps and check each intermediate result. (This is a good idea all the while you are learning APL.)

```
      char_mat_a←7 6⍴'Brown Green White Black SilverRed   Blue  '
```

Choose all the names that start with the letter B. The semicolon-one inside the brackets means the first column in all rows. Use the Boolean vector to designate which rows of the matrix you want, using the Compress function on the first dimension. Note that the backslash here is dyadic; hence it is the Compress function and not the Reduction operator.

```
      'B'=char_mat_a[;1]
1 0 0 1 0 0 1
      ('B'=char_mat_a[;1])⌿char_mat_a
Brown
Black
Blue
```

You can alphabetize the rows of the matrix easily.

```
       char_mat_a[⍋av4⍉char_mat_a;]
Black
Blue
Brown
Green
Red
Silver
White
```

This is just a peek at the immense power of APL. Whatever your data or your problem, you can do a huge amount of work quickly with APL. As you gain familiarity with the functions and syntax, you will discover untold riches of capability.

→

Chapter 3. Writing Your Own Functions

3.1 Programming in APL

You have already seen how much you can do in APL just using immediate execution mode. If you want to be able to perform calculations or actions repeatedly with different arguments or data, you can write your own functions. You are not restricted to the "program" format of other languages. You can write a function to do as much or as little as you want it to, and then use that function either standalone or in other statements.

For example, if you write a function named my_calc, you can simply execute the function and have the system display the result in the session, just as it displays the result of the statement 1+1 when you execute it.

Or you can assign the result of your function and use the variable with the result. You might type res ← my_calc, and then use the variable res in a statement such as b ← res+1.

You can also use your function in another statement, such as b← (my_calc+1) *2. Additionally, you can call your function from another function, so that a second function you write uses my_calc internally.

You can use a function to perform a small calculation with arguments you supply; You can use a function like you would use a subroutine in another language. Or you can use a function to encompass an entire program. It's in your hands.

3.2 Defining a Function

You define certain key information in the first line of your function, which we call the header. A function can be dyadic (two arguments), monadic (one argument) or niladic (no arguments). You also determine whether or not it returns a result. And you give the function its name. The name is the only required object in the header.

You invoke the function editor for a new function by typing)edit followed by a space and the del (∇) symbol (which is Alt+G on the keyboard), followed by a name (no space); for example)edit ∇my_calc. You can save your work in memory by typing Ctrl+E. If you want to go back to the function editor, just type)edit followed by a space and the name you assigned the function:
)edit my_calc

Note that this does not save the function permanently. If you want to save your work to disk, you must save the workspace by typing)save.

The header is line zero in the function. It contains from one to four names, three of which are "dummy" variable names. They exist only within the function and are, in essence, placeholders for the variables or values you will use when you actually invoke the function. The syntax is as follows:

```
[0] result← left_arg  fn_name  right_arg
```

The function name is the only required element. If you do not want your function to return an explicit result, you can omit the result variable and the left arrow. If you do not want a left argument, you can omit that name. If you put two names in the header (without the assignment arrow), the first one is the function name and the second is the single (right) argument. If you put three names, the middle one is the function name. You can also have a result with a function that has one or zero arguments. When you are ready to run the function, you simply type its name in the session with values for whatever arguments the function requires.

Note that having one dummy variable name as an argument does not limit you to one value. A single right argument can be a vector or matrix of values. You can use this array in calculations or you can put various values in it that you

use separately. For example, you could have a right argument of three values representing interest rate, term, and loan amount. In your function, you can assign the three values to separate variables:

```
[0] result ← calc_payment triple_argument
[1] interest ← triple_argument[1] ÷ 12
[2] loanlength ← triple_argument[2] × 12
[3] loanamount ← triple_argument[3]
```

This function supposes that you want a monthly payment. You divide the value input for interest by 12 to get a monthly value and multiply the value for term (assuming years) by 12 to get the number of months. Then, the payment calculation is well known:

```
[4] result ← (loanamount × interest) ÷ (1 - (1 ÷ 1 + interest) * loanlength)
```

In the session, you invoke the function by typing:

```
      calc_payment .075 30 50000
```

Or you could assign a variable, t←.075 30 50000, and invoke the function with:

```
      calc_payment t
```

Once you have a function like calc_payment, you can change the values for interest, term, and/or amount and see the effect on the monthly payment as fast as you can type the numbers and press Enter.

3.3 Localization

It is an important concept in APL that the variables in a function header are placeholders only for as long as the function is active. This is called localization. In the example above, the variable named 'result' does not exist after the function completes. You have a statement in your function that assigns the result to this variable. If you just invoke the function, the system displays the answer in the session. You can also assign the result to a variable when you invoke the function.

```
      payment ← calc_payment .075 30 50000
```

If you do this, the variable named payment holds the answer. You can then use this variable for further calculations in your session.

If you want to localize variables other than the result and arguments, you can place their names in the header following the argument, separated by semicolons. For example, the header above might look like this.

```
[0] result ← calc_payment triple_argument;interest;loanlength;loanamount
```

The value of doing this is twofold. It is tidy, in that no extraneous variables are created in the workspace; and, it avoids unintended side effects where you change something in the session. Typically, you might use short variable names. You could write every function you ever wanted using the same variables, and create no conflicts as long as the variables are localized.

```
[0] z← x fn_name y;r;s;t
```

As long as you use only r, s, t, x, y, and z in your function, you never create a conflict, even if one function calls another.

```
[0] z← x fn_name_a y;r;s;t  ⍝ This is the header for fn_name_a
. . .
[4] r ← s fn_name_b t       ⍝ This statement calls another function.

[n] z←    ⍝ Calculation in function fn_name_a using r

[0] z← x fn_name_b y;r;s;t  ⍝ This is the header for fn_name_b
. . .
[m] z←   ⍝ Calculation in function fn_name_b using the values s and t within
         ⍝ fn_name_a as arguments; these do not affect the use of s or t here.
```

Note that while you CAN use single letters as variable names, we recommend that you use descriptive names. You will find it much easier to maintain or change your code when your variable names provide information.

3.3 Program Flow -- Labels and Branches

Unless you specify otherwise, the system executes a function line by line until it reaches the end. If you want to control the flow of execution within your function, you can use the right arrow (→) to branch. You get the right arrow from the keyboard immediately to the right of the left arrow, using Alt+ the right bracket key.

If you branch to zero, you exit the function: [n] →0 You can branch to a line number, but that isn't very good programming style, and it makes it hard to modify your function. So, you can branch to a label. You choose an arbitrary name for your label, and place the label followed by a colon (no space) on a line of your function. Then you can branch to that line from anywhere in the function, and the system begins execution on the next line. For example:

```
[0]  my_sample_fn x;y;z
[1]  . . .

[m+1]  → initialize
[m+2]  continue:
[m+3]  . . .

[n-1] →0
[n]  initialize:
[n+1] y←x+1
[n+2] z←(x+y)*2
[n+3] → continue
```

In this example, the branch at line [m+1] sends the program to line [n], where some variables are initialized. At line [n+3] the program branches back to line [m+2], whence it continues up to line [n-1]. At that point the function exits. You might do this for convenience of reading your function. The initialization process is at the end and out of the way of analyzing the main thrust of the function.

You can also have conditional branches, using the Compress function (dyadic /). In this case you can construct a test, using a Boolean function, that returns a one or a zero. If the result is one, the function branches; otherwise it continues. You can use this technique to loop a number of times through a portion of your function, for example.

```
[0]  loop_fn
[1]  i←0
[2]  begin:
[3]  i ← i+1
[4]  . . .

[n]  →(i≤10)/begin
```

The above example executes from line [3] through line [n] with the variable i having the value 1. At line [n], it loops back to line [2], and continues to execute until the variable i has a value greater than 10. When i becomes greater than 10, the expression in parentheses is not true; therefore, it returns zero, the branch is not performed, and the function ends because there is nothing beyone line [n] to execute.

3.4 Conclusion

Other than branches, writing your own functions is very much like writing APL in immediate execution. It simply packages a number of steps under the name you assign and executes the steps as a unit.

This ends the instructional part of the documentation for APLSE. Chapters 4, 5

and 6 contain summary descriptions of all the basic APLSE functions. As you can tell from this very brief introduction, APL has a richness of capability unmatched by ordinary programming languages. As you use APL, you will discover there are easy ways to do just about anything you want to do to manipulate numeric or character data. Rounding numbers is done with ⌊n+.5, for example.

There are many areas that are not even touched on here. You can save data in files. You can make changes in the computing environment. For example, if you want all your sequences to start at zero instead of one, there is a system variable, Index Origin, that you can set: ⎕io←0

There are a number of organizations devoted to APL in the United States and around the world. Some of these organizations produce regular publications to which you can subscribe. There is a vast body of knowledge and technique that you may discover in talking with someone who has used APL to actually build applications.

If this rudimentary system interests you, Manugistics, Inc., produces full-scale versions of APL for various platforms and operating systems, including Unix, DOS, and Windows. These products go under the name APL*PLUS.

Commercial versions include sophisticated memory management, capacity for large variables, many utilities and interfaces, and telephone support options. Call (800) 592-0050 (in MD, (301) 984-5123; from outside the U.S., (301) 984-5412) for more information about these advanced systems. Manugistics does not support APL*PLUS SE, but a BBS forum is available by dialing (301) 984-5222 (full duplex, up to 14.4Kbps,n,8,1).

→

Chapter 4. Function Reference -- Arithmetic Functions

Function descriptions below are given in the following format:

```
⊛  Name   ¦  symbol      ◊   Definition in words
   Syntax                ⍝   Restrictions on arguments, if any
      Example (starting at six-space indent)
Result
```

Examples are usually given with vector arguments for simplicity and compactness. Functions generally work with arrays of more dimensions.

Note: The word "conforming" when applied to the arguments of a dyadic function means that the two arguments must agree in some manner; often, they must have the same shape. When one argument is a scalar, the system extends it to match the shape of the other argument.

res means result; arg means the argument to a monadic function; larg means left argument and rarg means right argument to a dyadic function.

```
⊛  Conjugate  ¦  monadic +   ◊  Return the value of arg
   res ← + arg               ⍝  where arg is any numeric array
      + 6  18.2  ¯5
6 18.2 ¯5
```

```
⊛  Plus   ¦   dyadic +       ◊  Add larg to rarg
   res ← larg + rarg         ⍝  larg, rarg conforming numeric arrays
      ¯3 2 1 + 6 18.2 ¯5
3 20.2 ¯4
```

```
⊛  Negate  ¦  monadic -      ◊  Change the sign of arg
   res ← - arg               ⍝  any numeric array
      - 6 18.2 ¯5
¯6 ¯18.2 5
```

```
⊛  Minus  ¦  dyadic -        ◊  Subtract rarg from larg
   res ← larg - rarg         ⍝  larg, rarg conforming numeric arrays
      ¯3 2 1 - 6 18.2 ¯5
¯9 ¯16.2 6
```

```
⊛  Signum   ¦  monadic ×     ◊  Return the sign of arg
   res: 1 if arg is positive; 0 if arg is zero; ¯1 if arg is negative
                             ⍝  arg can be any numeric array
      × 6 18.2 ¯5 0
1 1 ¯1 0
```

```
⊛  Times   ¦   dyadic ×      ◊  Multiply larg by rarg
   res ← larg × rarg         ⍝  larg, rarg conforming numeric arrays
      ¯3 2 1 × 6 18.2 ¯5
¯18 36.4 ¯5
```

```
⊛  Reciprocal ¦ monadic ÷    ◊  Return the reciprocal of arg (1 divided by arg)
   res ←  ÷ arg              ⍝  any non-zero numeric array
      ÷ 6 18.2 ¯5
0.1666666667 0.05494505495 ¯0.2
```

```
⊛  Divide  ¦  dyadic ÷       ◊  Divide larg by rarg
   res ← larg ÷ rarg         ⍝  larg, rarg conforming numeric arrays;
                               rarg must be non-zero except 0÷0 returns 1.
      ¯3 2 1 0 0 ÷ 6 18.2 ¯5  ¯1 0
¯0.5 0.1098901099 ¯0.2 0 1


⊛  Exponential ¦ monadic *   ◊  Raise the value e (2.71828...) to the power arg
   res ← * arg               ⍝  any numeric array
      * 6 18.2 ¯5
403.4287935 80197267.41 6.737946999E¯3


⊛  Power  ¦  dyadic *        ◊  Raise larg to the power rarg
   res ← larg * rarg         ⍝  larg, rarg conforming numeric arrays
      6 18.2 ¯5 * ¯3 2 1
4.62962963E¯3 331.24 ¯5


⊛  Natural log ¦ monadic ⍟   ◊  Compute the natural logarithm (base e) of arg
   res ← ⍟ arg               ⍝  any positive numeric array
      ⍟ 3 2.7182818284 1
1.098612289 1 0


⊛  Logarithm  ¦  dyadic ⍟    ◊  Compute the logarithm of rarg to the base larg
   res ← larg ⍟ rarg         ⍝  larg, rarg conforming positive numeric arrays
      2 49 4 ⍟ 8 7 0.25
3 0.5 ¯1


⊛  Ceiling  ¦  monadic ⌈     ◊  Round up to the nearest integer
   res ← ⌈ arg               ⍝  any numeric array
      ⌈ 3.14159 ¯1.5 6
4 ¯1 6


⊛  Maximum  ¦  dyadic ⌈      ◊  Select the larger of two numbers
   res ← larg ⌈ rarg         ⍝  larg, rarg conforming numeric arrays
      6 18.2 ¯5 ⌈ 3.14159 19.3 ¯4.9
6 19.3 ¯4.9


⊛  Floor  ¦  monadic ⌊       ◊  Round down to the nearest integer
   res ← ⌊ arg               ⍝  any numeric array
      ⌊ 3.14159 ¯1.5 6
3 ¯2 6


⊛  Minimum  ¦  dyadic ⌊      ◊  Select the smaller of two numbers
   res ← larg ⌊ rarg         ⍝  larg, rarg conforming numeric arrays
      6 18.2 ¯5 ⌊ 3.14159 19.3 ¯4.9
3.14159 18.2 ¯5


⊛  Magnitude  ¦  monadic |   ◊  Return the absolute value of arg
   res← | arg                ⍝  any numeric array
      | 6 18.2 ¯5
6 18.2 5


⊛  Residue  ¦  dyadic |      ◊  Find the remainder after dividing larg by rarg
   res ← larg | rarg         ⍝  larg, rarg conforming numeric arrays
      3 3 ¯3 ¯3 1 ¯1 0 | 7.2 ¯7.2 7.4 ¯7.4 7.6 ¯7.6 8.5
1.2 1.8 ¯1.6 ¯1.4 0.6 ¯0.6 8.5
```

Note: 0 | n returns n; be careful with negative numbers in either argument
The function explicitly is r - (⌊ r ÷ l) × l

⊛ Factorial ¦ monadic ! ◊ Compute the factorial of arg
res ← ! arg ⍝ any numeric array, except negative integers

```
   ! 0 5 2.5 ¯2.5
1 120 3.32335097 2.363271801
```

Note: ! 0 returns 1; for positive integers res is the product of ⍳ arg;
all fractional numbers are computed using the gamma function on arg+1

⊛ Binomial ¦ dyadic ! ◊ Find the number of combinations of rarg objects
selected larg at a time
res ← larg ! rarg ⍝ larg, rarg conforming positive numeric arrays

```
   1 2 5 2.5 ! 5
5 10 1 10.86497745
```

⊛ Roll ¦ monadic ? ◊ Select a pseudorandom integer from the set ⍳ arg
res ← ? arg ⍝ any positive integer array

```
   ? 1995 18 3
1355 13 3
```

⊛ Deal ¦ dyadic ? ◊ Select larg pseudorandom integers without
duplicates from the set ⍳ rarg
larg ? rarg ⍝ larg, rarg non-negative integer scalars larg≤rarg

```
   8 ? 10
1 5 3 4 9 6 8 7
```

Note: Deal produces no duplicate values in the result; if you want to
maintain equal odds of selecting any number use (? larg ⍴ rarg).

⊛ Base value ¦ dyadic ⊤ ◊ Convert arg to the number system defined by larg
res ← larg ⊤ rarg ⍝ larg, rarg conforming numeric arrays

```
   7 24 60 ⊤ 2 2 2
3002
```

Note: This example converts 2 days, 2 hours, 2 minutes to minutes.

⊛ Representation¦dyadic ⊥ ◊ Represent rarg in the number system defined by larg
res ← larg ⊥ rarg ⍝ larg, rarg any numeric arrays

```
   0 5 8 ⊥ 666
16 3 2
```

Note: This example converts hours to work weeks, work days and hours.

⊛ Matrix inverse ¦ monadic ⌹ ◊ Calculate the inverse of a matrix
res ← ⌹ arg ⍝ any numeric scalar, vector, or matrix.
Note: If arg is nonsingular, but not square, it must have more rows than
columns; res is then the least squares approximation to the inverse.

```
   ⌹ 2 2 ⍴ 1 1 2 3
 3 ¯1
¯2  1
```

⊛ Matrix divide ¦ dyadic ⌹ ◊ Solve a set of simultaneous equations
res ← larg ⌹ rarg ⍝ larg, rarg numeric scalars, vectors or matrices

```
                                rank (# of dimensions) or rarg ≥ rank of larg
   Note:  If rarg is a matrix and not square, it must have more rows than cols.
      14 26 ⌹ 2 2 ⍴ 1 3 4 2
5 3

    This example returns x, y where x+3y=14 and 4x+2y=26


⊛  Pi times ¦ monadic ○     ◊  Multiply the value pi (3.14159...) by arg
   res ← ○ arg              ⍵  any numeric array
      ○ 0 1 2
0 3.141592654 6.283185307

   Note:  This symbol is the circle, Alt+O (letter O).


⊛  Trigonometric functions ¦ dyadic ○  ◊  see below
   res ← larg ○ rarg         ⍺  larg, integer array ¯7 ≤ larg[⍳] ≤ 7
                                rarg, valid numeric array measured in radians
   Note:  larg specifies which of 15 functions is calculated.
     1 SIN       5 SINH       ¯1 ARCSIN       ¯5  ARCSINH
     2 COS       6 COSH       ¯2 ARCCOS       ¯6  ARCCOSH
     3 TAN       7 TANH       ¯3 ARCTAN       ¯7  ARCTANH
     4 (1+ rarg*2) *.5        ¯4 (¯1+ rarg *2) *.5        0  (1- rarg*2) *.5

      1○ 1.5708
1
      ¯1○ 1
1.570796327
      0○ .8
0.6
      4○ 2.4
2.6
      ¯4○ 2.6
2.4
→
```

Chapter 5. Function Reference -- Structural Functions

The functions described in this chapter allow you to manipulate data in ways other than arithmetic calculations. With these functions, you can arrange or rearrange data in arrays, select subsets of your data in arrays, or a combination of these actions.

Most of these functions work on either numeric arrays or character arrays. The examples usually use numeric arrays; you can experiment with character arrays to see the effects. Examples also are usually given with vector arguments; these functions generally work with arrays of more dimensions.

Examples are usually given with small arguments for simplicity and compactness. Several examples may be strung across the page to save space, using the pound sign (#) as a separator. APL would not literally provide the output as shown.

```
⊛  Name   ¦  symbol      ◊   Description in words
   Syntax                ⍝   Restrictions on arguments, if any
   Explanation of the result (if necessary)
      Example (starting at six-space indent)
Result
```

res means result; arg means the argument to a monadic function; larg means left argument and rarg means right argument to a dyadic function.

```
⊛  Index generator ¦ monadic ⍳  ◊  Return the set of integers up to arg
   res ← ⍳ arg                  ⍝  any positive integer scalar
      ⍳ 5
1 2 3 4 5
```

```
⊛  Index   ¦   dyadic ⍳         ◊  Find the location of items in an array
   res ← larg ⍳ rarg            ⍝  larg, any vector; rarg, any array
   ⍝ res is the index of the first occurrence in larg of each item of rarg.
   ⍝ If larg does not contain the item, the item in res is one greater than the
   ⍝ length of larg.
      1 2 3 4 3 2 1 ⍳ (2 2 ⍴2 4 5 3)
2 4
8 3
```

```
⊛  Shape  ¦  monadic ⍴    ◊  Return the shape (length of each dimension) of arg
   res ← ⍴ arg            ⍝  any array
   ⍝  Note:  The shape of a scalar is blank (not zero).  A variable with zero
   ⍝         shape is an empty vector.
       ⍴ 1 2 3      #      ⍴ 'abc'    #      ⍴  99         #      ⍴⍬
3                   #3                #                    #0
```

```
⊛  Reshape  ¦  dyadic ⍴   ◊  Create an array of specific shape
   res ← larg ⍴ rarg      ⍝  larg, integer scalar or vector; rarg, any array
   ⍝  res is the items of rarg selected in order and formed into the shape
   ⍝  specified by larg.  Some elements of rarg may be unused or duplicated.
      2 4 ⍴ 1 2 3
1 2 3 1
2 3 1 2
```

```
⊛  Ravel  ¦  monadic ,   ◊  Change an array into a vector
   res ← , arg           ⍝  any array
   ⍝  res is all the items of arg in the same order as arg, but as a vector
      , 2 4 ⍴ ⍳3
1 2 3 1 2 3 1 2
```

⊛ Catenate ¦ dyadic , or ⍪ ◊ Join two arrays along a specified axis
res← larg ⍪rarg ◊ res← larg ,rarg ◊ res← larg ,[n]rarg ◊ res← larg ⍪[n]rarg

res contains all the items of larg and all the items of rarg; the shape of res depends on how you specify the function. If you specify catbar (⍪), the arrays are joined along the first dimension; if you use the comma (,), the arrays are joined along the last dimension. If you explictly specify an integer axis in brackets, the arrays are joined along that axis. If you specify a fractional value in brackets, the system creates a new dimension in between the integers nearest to [n].

larg and rarg can be any arrays whose dimensions match along each axis other than the one specified; or either larg or rarg can be a scalar.

```
   a←1 2 3
   b←4 5 6
   a , b          #      a ,[1.5] b      #      7 8 9 ⍪ a,[.5]b
1 2 3 4 5 6       # 1 4                  # 7 8 9
   a ,[.5] b      # 2 5                  # 1 2 3
1 2 3             # 3 6                  # 4 5 6
4 5 6
```

⊛ Reverse ¦ monadic ⊖ or ⌽ ◊ Reverse the order of arg along a specified axis
res ← ⊖ arg ◊ res ← ⌽ arg ◊ res← ⊖[i] arg ◊ res ← ⌽[i] arg
⍝ If you use rotate-bar (⊖), the system reverses along the first dimension.
⍝ If you use rotate (⌽), it reverses along the last dimension (columns).
⍝ If you specify an axis, [i] must be an integer scalar.

```
   a←2 3⍴⍳6
   ⌽a              #      ⊖a          #      ⌽'able was I'
3 2 1              # 4 5 6            #I saw elba
6 5 4              # 1 2 3
```

You can experiment with specifying the axis with arrays of more dimensions.

⊛ Rotate ¦ dyadic ⊖ or ⌽ ◊ Shift the elements of rarg along a specified axis in an amount designated by larg
res← larg ⊖rarg ◊ res← larg ⌽rarg ◊ res←larg ⊖[i]rarg ◊ res← larg ⌽[i]rarg
⍝ larg, integer scalar or vector of length matching specified dimension
⍝ of rarg; rarg, any array; i, non-negative integer scalar

```
  arg←3 3⍴⍳9
   1 ⊖ arg       #      ¯1 ⌽ arg       #      1 2 3 ⌽ arg
4 5 6            #3 1 2                #2 3 1
7 8 9            #6 4 5                #6 4 5
1 2 3            #9 7 8                #7 8 9
```

You can experiment with specifying the axis with arrays of more dimensions.

⊛ Transpose ¦ monadic or dyadic ⍉ ◊ Reverse or reorder the axes of an array
res← ⍉ arg ⍝ arg, any array
res← larg ⍉ rarg ⍝ rarg, any array; larg, non-negative integer vector with length equal to rank (# of dimensions) of rarg

```
   arg←2 3⍴ ⍳6   #  If rarg← 2 3 4⍴⍳24 then the shape of the result is:
   ⍉ arg         #  statement              shape of result (⍴ res)
1 4              #      ⍉ rarg                   4 3 2
2 5              #      1 3 2 ⍉ rarg             2 4 3
3 6              #      3 2 1 ⍉ rarg             4 3 2
```

You can also experiment with dyadic transpose, using left arguments that use integers starting with 1 and not skipping any numbers, but that repeat some numbers and therefore do not specify every dimension.

```
⊛  Take  ¦  dyadic ↑        ◊  Select a set of elements from an array
   res ← larg ↑ rarg        ⍝  rarg, any array; larg, integer scalar or vector
   ⍝  with one element for each dimension of rarg.  You can take from the
   ⍝  beginning of rarg with positive larg or from the end of rarg with
   ⍝  negative larg.  You can overtake, that is, you can specify more elements
   ⍝  than exist in rarg.  In this case, the system pads the result with zeroes
   ⍝  for numeric rarg or with blanks for character rarg.  The result will
   ⍝  always have the shape of the absolute value of the left argument.
   2 ↑ ⍳6        #      ¯2 ↑⍳6        #     4 ↑⍳3    #    2 ¯2 ↑ 3 3 ⍴ ⍳9
1 2              #5 6                 #1 2 3 0       # 2 3
                                                      # 5 6

⊛  Drop  ¦  dyadic ↓      ◊  Exclude a set of elements from an array
   res ← larg ↓ rarg        ⍝  rarg, any array; larg; integer scalar or vector
   ⍝  with one element for each dimension or rarg.  You can drop from the
   ⍝  beginning of rarg with positive larg; you can drop from the end of rarg
   ⍝  with negative larg.  If you specify more elements than exist in rarg, the
   ⍝  result is an empty array.
   2 ↓ ⍳6        #      ¯2 ↓⍳6        #      4 ↓ ⍳3
3 4 5 6          #1 2 3 4             #

⊛  Expand  ¦ dyadic ⍀ or \    ◊  Expand rarg along a specified dimension, padding
                                 with blanks or zeroes where larg contains zeroes
   res← larg ⍀rarg ◊ res← larg \rarg ◊ res← larg ⍀[i]rarg ◊ res← larg \[i]rarg
   ⍝  rarg, any array; larg consists of 1s and 0s, and must have as many 1s as
   ⍝  rarg has elements along the specified dimension.
                        #  lettermat← 2 3 ⍴ 'abcdef'
      0 0 1 0 1 \ 2 3   #      1 0 1 ⍀ lettermat   #      1 0 1 0 1 \ lettermat
0 0 2 0 3               # abc                     #a b c
                        #                       #d e f
                        # def

   The backslash symbols are also used for the Scan operator.  See Chapter 6.


⊛  Replicate  ¦  dyadic ⌿ or /  ◊ Select the elements of rarg along a specified
                                   dimension as many times as designated by larg
   res← larg ⌿rarg ◊ res← larg /rarg ◊ res← larg ⌿[i]rarg ◊ res← larg /[i]rarg
   ⍝  rarg, any array; larg, non-negative integer scalar or vector with as many
   ⍝  elements as rarg has along the specified dimension.
      1 2 3 /lettermat   # 1 2⌿ lettermat    #      1 0 2 0 3 / 1 2 3 4 5
abbccc                  # abc               # 1 3 3 5 5 5
deefff                  # def
                        # def

   Note:  The Replicate function is also called Compress; if you use a Boolean
          left argument (1s and 0s), you select only those elements from larg
          that you want and squeeze out the others.

  The slash symbols are also used for the Reduction operator.  See Chapter 6.


⊛  Numeric Grade Up  ¦ monadic ⍋  ◊  Return the ascending sort order of arg
   res ← ⍋ arg                    ⍝  any numeric array
   ⍝  This function works along the first dimension if the rank of arg ⩾ 2.
   ⍝  In the example below, the result shows the first element of rarg in
   ⍝  ascending order is the fourth element.
      ⍋ 9 3 7 1 5
4 2 5 3 1

   Note:  If you want to sort an array, use this index to reassign the items.
      vector← 5 8 3 8 4 2 5
      ⍋vector
6 3 5 1 7 2 4
```

```
      vector ← vector[⍋vector]
      vector
2 3 4 5 5 8 8
```

```
⊛  Numeric Grade Down  ¦ monadic ⍒  ◊  Return the descending sort order of arg
   res ← ⍒ arg                    ⍝  any numeric array
   ⍝  This function works along the first dimension if the rank of arg ⩾ 2.
   ⍝  In the example below, the first element of rarg in descending order is in
   ⍝  the first position, and the second element in descending order is third.
      ⍋ 9 3 7 1 5
1 3 5 2 4
```

```
⊛  Character Grade Up  ¦  dyadic ⍋  ◊  Return the ascending sort order of rarg
                                       according to the scheme defined by larg
   res ← larg ⍋ rarg               ⍝  rarg, larg any character arrays
      x ← 'A quick brown fox jumps over the lazy dog.'
      x[⎕AV⍋x]
        .Aabcdeefghijklmnoooopqrrstuuvwxyz

   Note that the spaces are characters and they come at the beginning of res.
   You can specify any set of characters in the left argument.  Typically you
   might choose a collating sequence such as larg←'AaBbCc' etc., or
   larg←'abc. . .','ABC . . .'  If characters appear in rarg that do not appear
   in larg, the system puts them at the end of the sequence in their rarg order.
```

```
⊛  Without  ¦  dyadic ~     ◊  Select the elements of larg that are not in rarg
   res ← larg ~ rarg        ⍝  larg, any scalar or vector; rarg, any array
     'beekeeper'~'e'  #     1 2 3 4 5 ~ 2 4  #     'paragon'~'paragon'~'aeiou'
bkpr                  #1 3 5                 # aao
→
```

Chapter 6. Function Reference -- General Functions and Operators

The functions described in this chapter do not fall in the category of either arithmetic or structural functions. Many of them allow you to compare or test data; with the results of the test, you can use structural and arithmetic functions to advantage. There is also a category of APL symbols called operators which allow you to modify the way functions work. In addition, some uses of special symbols that are neither functions nor operators are described.

Examples are usually given with vector arguments; these functions generally work with arrays of more dimensions. Examples also usually use numeric arrays. You can experiment with character arrays to see the effects.

Examples are usually given with small arguments for simplicity and compactness. Several examples may be strung across the page to save space, using the pound sign (#) as a separator. APL would not literally provide the output as shown.

```
⊛  Name   ¦  symbol      ◊   Description in words
   Syntax                ⍝   Restrictions on arguments, if any
   Explanation of the result (if necessary)
      Example (starting at six-space indent)
Result
```

res means result; arg means the argument to a monadic function; larg means left argument and rarg means right argument to a dyadic function.

6.1 Boolean Functions

Boolean arrays consist of only two values, 0 and 1. You can think of these values as representing the absence or presence of a characteristic, no/yes, or false/true. A function that returns a Boolean array describes a relationship between the data arrays that are its arguments. The arrays that are arguments can be of other data types.

6.1.1 Functions that Return a Boolean Array

```
⊛  Equal  ¦  dyadic =      ◊  Return 1 if larg equals rarg
   res ← larg = rarg       ⍝  larg, rarg any conforming arrays
      'I' = 'MANUGISTICS'     #      1 2 3 = 2 1 3
0 0 0 0 0 1 0 0 1 0 0         #0 0 1

⊛  Not equal  ¦  dyadic ≠  ◊  Return 1 if larg does not equal rarg
   res ← larg ≠ rarg       ⍝  larg, rarg any conforming arrays
      'statistics' ≠ 'satisfying'        #      1 2 3 ≠ 2 1 3
0 1 1 1 1 1 1 0 1 1                      #1 1 0

⊛  Greater than  ¦  dyadic >  ◊ Return 1 if larg is greater than rarg
   res ← larg > rarg       ⍝  larg, rarg conforming numeric arrays
      1 2 3 > 2 1 3
0 1 0

⊛  Greater than or equal  ¦  dyadic ≥  ◊  Return 1 if larg ≥ rarg
   res ← larg ≥ rarg       ⍝  larg, rarg, conforming numeric arrays
      1 2 3 ≥ 2 1 3
0 1 1

⊛  Less than  ¦  dyadic <  ◊ Return 1 if larg is less than rarg
   res ← larg < rarg       ⍝  larg, rarg conforming numeric arrays
      1 2 3 < 2 1 3
1 0 0

⊛  Less than or equal  ¦ dyadic ≤ ◊ Return 1 if larg less than or equal to rarg
```

```
   res ← larg ⩽ rarg              ⍝  larg, rarg, conforming numeric arrays
      1 2 3 ⩽ 2 1 3
1 0 1

⊛  Match  ¦  dyadic ≡      ◊  Return 1 if larg and rarg have the same rank,
                                 shape and values
   res ← larg ≡ rarg       ⍝  larg, rarg any arrays
      1 2 3 ≡ 2 1 3       #      'xyzzy' ≡ 1 5 ⍴ 'xyzzy'  #     'xyzzy'≡'xyzzy'
0                          #0                               #1

⊛  Member of  ¦  dyadic ∈  ◊ Return 1 if rarg contains an element of larg
   res ← larg ∈ rarg       ⍝  larg, rarg any arrays
   ⍝  res has the shape of larg;  the system looks for single elements of larg
      'abc' ∈ 'banana'
1 1 0

⊛  Find  ¦  dyadic ⊆     ◊  Return 1 where larg starts if rarg contains larg
   res ← larg ⊆ rarg     ⍝  larg, rarg any arrays
   ⍝  res has the shape of rarg; the system looks for the entire array larg
      'ana' ⊆ 'banana'
0 1 0 1 0 0


6.1.2  Logical Functions

A Logical function uses only Boolean arrays as arguments and also return a
Boolean array as its result.  You can also use Boolean arrays as arguments to
non-Boolean functions.  The results may not be Boolean.  For example, you can
sum a Boolean vector to find the number of occurrences.

⊛  AND  ¦  dyadic ∧       ◊  Return 1 if both larg and rarg are 1
   res ← larg ∧ rarg      ⍝  larg, rarg any conforming Boolean arrays
      0 0 1 1 ∧ 0 1 0 1
0 0 0 1

⊛  OR  ¦  dyadic ∨        ◊  Return 1 if either or both larg and rarg are 1
   res ← larg ∨ rarg      ⍝  larg, rarg any conforming Boolean arrays
      0 0 1 1 ∨ 0 1 0 1
0 1 1 1

⊛  NAND  ¦  dyadic ⍲      ◊  Return 0 if both larg and rarg are 1
   res ← larg ⍲ rarg      ⍝  larg, rarg any conforming Boolean arrays
      0 0 1 1 ⍲ 0 1 0 1
1 1 1 0

⊛  NOR  ¦  dyadic ⍱       ◊  Return 1 if neither larg nor rarg is 1
   res ← larg ⍱ rarg      ⍝  larg, rarg any conforming Boolean arrays
      0 0 1 1 ⍱ 0 1 0 1
1 0 0 0

⊛  NOT  ¦  monadic ~      ◊  Return 1 if arg is 0; return 0 if arg is 1
   res ← ~ arg            ⍝  arg, any Boolean array
      ~ 0 1 0 1
1 0 1 0

6.2  Execute and Format Functions

⊛  Execute  ¦  monadic ⍎  ◊  Execute an APL statement that is stored as a
                                 character vector
   res ← ⍎ arg             ⍝  arg, any character vector
      stv←'((I1*2)+I2*2)*.5'
      I1←3
      I2←4
      ⍎ stv
5
      ((I1*2)+I2*2)*.5
5
```

Note: You can use Execute to turn numerals (character representation of numbers) into numeric data, but it is preferable to use the system function ⎕FI.

```
⍝  Format  ¦  monadic ⍕    ◊  Represent numeric data in character form
   res ← ⍕ arg             ⍝  arg, any array, but characters are unchanged
      ⍕ 2 3 ⍴ ⍳ 6          #      'ab' ≡ ⍕'ab'
 1 2 3                     #1
 4 5 6
      ' '∈ ⍕ 2 3 ⍴ ⍳ 6
1 0 1 0 1 0
1 0 1 0 1 0
```

```
⍝  Pattern format  ¦  dyadic ⍕    ◊  Represent numeric data in character form,
                                        formatting the result according to larg
   res ← larg ⍕ rarg              ⍝  rarg, any numeric array; larg, integer
   ⍝ scalar, pair, or vector of integer pairs.  The first integer of a pair
   ⍝ specifies the column width; 0 requests a field large enough to hold the
   ⍝ largest number.  The second integer specifies the number of decimal places.
   ⍝ If the second number is negative, the system uses exponential notation.
   ⍝ If there is only one number, it defines the number of decimal places.
      1 0 4 1 6 2 ⍕ 2 3 ⍴ ⍳ 6   #      1 ⍕ 2 3 5         #       1 0 ⍕ 2 3 5
1 2.0  3.00                     #2.0 3.0 5.0             #235
4 5.0  6.00
```

6.3 Operators

Operators are a special group of APL symbols that modify functions. An operator takes a function (or two functions) as an operand, and the result is a derived function that acts on one or two arrays. Operators greatly expand the power of APL and allow complex manipulation of data with very concise expressions.

```
⍝  Reduction  ¦ one function (f) operand preceding / or ⌿
   res ← f ⌿ arg  ◊  res ← f / arg  ◊ res ← f ⌿[i] arg  ◊ res ← f /[i] arg
   ⍝  Apply the function across an array, eliminating the dimension specified
   ⍝  by i in the process.  The default dimension for ⌿ is the first dimension;
   ⍝  the default dimension for / is the last.  arg, any array valid for f.
      +/ ⍳6
21
      ×⌿2 3 ⍴ ⍳ 6
4 10 18
```

Note that the right to left execution of APL applies to this operation. If you have a statement like ÷/1 5 9, it is the same as writing 1 ÷ 5 ÷ 9. When APL executes this statement it divides 5 by 9 and then divides 1 by the intermediate result. You could express it as: res ← 1 ÷ (5 ÷ 9)

```
⍝  Scan  ¦  one function operand preceding \ or ⍀
   res ← f ⍀ arg  ◊  res ← f \ arg  ◊ res ← f ⍀[i] arg  ◊ res ← f \[i] arg
   ⍝  Apply successive reductions to the array along the specified dimension.
   ⍝  The default dimension for ⍀ is the first dimension; the default for \
   ⍝  is the last.  arg, any array valid for the function f.
      +\ ⍳6
1 3 6 10 15 21
      ×⍀ 3 2 ⍴ ⍳6
  1  2
  3  8
 15 48
```

Note that successive values incorporate more elements of the array from left to right, but the system calculates each value using right-to-left evaluation. Thus, the three successive values of ÷\1 5 9 are 1, 1÷5, and 1 ÷ (5÷9).

```
   ÷\1 5 9
1 0.2 1.8
```

```
⍣ Outer product  ¦  one function (g) following ∘.  ◊  applies to two arrays
  ⍝  Note that this compound symbol is jot-dot, made by Alt+J and the period.
  res ← larg ∘.g rarg
  ⍝  Apply the function between every possible pairing of items from larg and
  ⍝  rarg.  larg, rarg, any arrays valid for g.
      2 3 5 ∘.* 0 1 2 3
   1   2   4   8
   1   3   9  27
   1   5  25 125


⍣  Inner product  ¦  two functions separated by .  ◊  applies to two arrays
   res ← larg f.g rarg
   ⍝  Generalized matrix multiplication.  larg, rarg, arrays valid for f and g
   ⍝  where the last dimension of larg is equal to the first dimension of rarg.
   ⍝  This function applies the second function (g) between elements of the last
   ⍝  dimension of larg and corresponding elements of the first dimension of
   ⍝  rarg, followed by applying reduction using the first function (f).
      3 5 7 ×.- 1 2 3
24                    ⍝  This value is 2×3×4

      (2 3 ⍴ ⍳ 6) +.× 3 4 ⍴ ⍳ 12   ⍝  Matrix multiplication
  38  44  50  56
  83  98 113 128
```

6.4 Other Symbols Used by APL

APL uses the symbols below for purposes other than functions or operators.
These brief explanations provide the Symbol, Name, Use, Comments, and Example.

```
⍣  ¯  High Minus ¦ used to denote negative numbers
      ⍝ ¯2 is the result of 0 - 2



⍣  '  Single quote  ¦  used to delimit character strings
      ⍝  The variable 'abc' is a three-element character vector; so is '123'
      ⍝  If you want to include an apostrophe in a character string, double it.
      'Joe''s'
Joe's

⍣  ⍬  Zilde  (combination of zero and tilde)  ¦ the empty (numeric) vector
      ⍝  This is a placeholder that is not the same as zero.  It has shape 0.
      ⍬ ≡ ⍳0
1
      Note:  You can define an empty character vector with ''.

⍣  ←  Left arrow  ¦ assignment of values to a variable
      vec ← 3 + ⍳ 6
      vec
4 5 6 7 8 9

⍣  []  Brackets  ¦  indexing and indexed assignment
       vec[3]
6
       vec[5 6]←1 2
       vec
4 5 6 7 1 2

⍣  ;  Semicolon  ¦  used to separate dimensions of a matrix in indexing
       mat← 2 3⍴ ⍳ 6
       mat[2;1]        ⍝  This is row 2, column 1
4
       mat[1;1 3]←8 9
```

```
      mat
 8 2 9
 4 5 6
```

⍟ ∇ Del ¦ Function designator
     ⍝ If you type ∇ in the session, you begin editing a new function.
     ⍝ If you type a second del on a later line, you end the editing session.

⍟ ◊ Diamond ¦ Statement separator
      ⍝ Useful mainly in user-written functions to put more than one small
      ⍝ statement on a given line. The system executes statements from left
      ⍝ to right; each statement is evaluated from right to left.

```
      I← 5 ◊ J← 3 ◊ I-J
2
```

⍟ ⍝ Lamp ¦ Comment designator
      ⍝ Everything to the right of the lamp is not evaluated by the system
      ⍝ Useful mainly in user-written functions for internal documentation

⍟ → Right arrow ¦ Branch within a function
      ⍝ Used in user-written functions to change the flow of the function.
      ⍝ Typically, you branch to a label. Branch to zero exits the function.
      ⍝ The branch statement would look like this: [n] → here

⍟ : Colon ¦ Appended to a name to designate a label in a function
      ⍝ You place the label at the point to which you want to branch.
      ⍝ The branch arrow sends the program to that point.
      ⍝ The label statement would look like this: [23] here:

⍟ ⎕ Quad ¦ Allow a user-written function to request (usually) numeric input.
      ⍝ The input must be a statement that APL can evaluate. You can also use
      ⍝ quad to display the contents of a variable or an expression.

⍟ ⍞ Quote-quad ¦ Allow a user-written function to request character input.
      ⍝ The system treats any input from the keyboard as characters.
      ⍝ Example of a function using the above.

```
    ∇ myfn;aort;num;z
[1]   'How many random numbers (up to 99) do you want?'
[2]   num←⎕ ⌊ 99  ◊  z←?numρ100   ⍝  Comment:  Two statements on the same line.
[3]   z                 ⍝  This statement causes the random numbers to display.
[4]   'Do you want to add (Enter add) or multiply (Enter times)?'
[5]   aort←3↑⍞    ⍝  Make aort length three so the tests don't cause an error.
[6]   →(aort='add')/plus                ⍝  Branch
[7]   →(aort='tim')/product
[8]   →neither              ⍝  If the function gets to here, the user erred.
[9]  plus:
[10]  +/z                   ⍝  Do a plus reduction.
[11]  →0
[12] product:               ⍝  This label is the target of the second branch.
[13]  ×/z                   ⍝  Do a times reduction.
[14]  →0
[15] neither:
[16]  'You did not respond with a valid answer.'
    ∇
```

Note: You can find myfn in the INITIAL workspace.

    ! ! ! WARNING: USE THIS NEXT SYMBOL WITH CAUTION ! ! !
⍟ ⍫ Del-tilde ¦ Lock a function
      ⍝ This allows you to have a function that neither you nor anybody else
      ⍝ can subsequently alter. Before you do this, make a copy of the
      ⍝ function you are about to lock so you do not have to recreate your
      ⍝ work from scratch.

```
⍝  ()  Parentheses  ⍝  used to group items in an expression
       ⍝  The system evaluates the expression within parentheses before
       ⍝  continuing with the normal right-to-left progression.   You can nest
       ⍝  parentheses beyond the level you could understand the nesting.
      (9-(8-(7-(6-(5-(4-(3-(2-1)))))))   ⍝  Same as normal evaluation
5
      9-8-7-6-5-4-3-2-1
5
      (((((((9-8)-7)-6)-5)-4)-3)-2)-1
¯27

⍝  Δ ⍙ _  ⍝  Delta, Delta-underscore, and Underscore
      ⍝  You can use these three symbols in the names of APL variables and
      ⍝  functions.  An underscore may not be the first character.
      Δvar← 5
      vec⍙← 1 2 3
      mat_2 ← 3 3 ⍴ ⍳ 9
      mat_2 ⍪ vec⍙ × Δvar
 1  2  3
 4  5  6
 7  8  9
 5 10 15
→
```