What is APL?

When you have installed the IBM 370 simulator with the APL source code and have installed the correct APL font, we are ready to go.

First a check:

```
      ↑↓←→
```

The box should contain 4 arrows, pointing up, down, left, and right. If not, check your font setup.

When you start the simulator two terminal windows open. Log in as the operator (314159) in any of them, load the OPFNS workspace, set the date, and create yourself as a user, e.g. 1001. Please see Jürgen's instructions for doing this.

In the other terminal log in as the user:

```
)1001
009) 10.00.42 03/07/14 MK

   A P L \ 3 6 0
```

You are now in execution mode, and can immediately calculate expressions:

```
      2+2
4
```

Note: The machine writes the answer to the left, and indents 6 spaces when it is your turn to write.

The terminals used for APL\360 were typewriter-type terminals, an IBM Selectric typewriter with a special APL typeball. Thus there were no way of correcting type mistakes on the paper. There's no cursor keys. The backspace key is used to write overstrikes: The exclamation sign ! is written: quote, backspace, and decimal point. To correct a mistake, backspace to the mistake and press shift-x. The machine writes an arrow below and let you type the rest of the line:

```
      2+3×4+5
         ∨
         +4+5
14
```

Some more examples:

```
      3×4
12
      4×5+6
44
      1÷3
0.3333333333
      4−7
¯3
```

WHAT? 4*5+6 should be 26, not 44? No, APL evaluates from the right, and all operators have the same priority. So the result is 4*(5+6) = 44. You can use parentheses to change this behaviour.

Note that negative numbers are marked with the ¯ symbol. The symbol – is a monadic operator working on everything to the right of it.

Please see Table 3.2 in the User's Manual for more scalar functions in APL.

We can assign the result of an expression to a variable:

```
      A←3+4
      A
7
```

The scalar functions also work with vectors of numbers, element-by-element:

```
      2 3 5×7 9 13
14  27  65
```

One of the parameters may be a scalar:

```
      1÷1 2 3
1  0.5  0.3333333333
```

The leftmost number 1 is just repeated to match the length of the right-hand-side.

Use the monadic iota (⍳) operator to generate a list of integers:

```
      ⍳10
1  2  3  4  5  6  7  8  9  10
      3×⍳10
3  6  9  12  15  18  21  24  27  30
```

The iota function generates the first N indices, and indices normally start from 1 in APL\360. This can be changed to 0, but let's forget about that.

Use the dyadic iota to find numbers, (*HAYSTACK*⍳*NEEDLE*):

```
      1 2 3 4⍳2 3 5 7 11 13
2  3  5  5  5  5
```

If the number to search for does not exist, the length of the vector to the left plus 1 is returned.

Vectors are one dimensional lists of numbers, but we can also work in higher dimensions. Use the shape operator rho (⍴) to generate a matrix:

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

We can also make a three-dimensional table:

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

The planes are printed seperated by a blank line.

The monadic shape function returns the dimension:

```
      A←3
      B←,4
      C←⍳10
      D←3 3⍴⍳9
      ⍴A

      ⍴B
1
      ⍴C
10
      ⍴D
3  3
```

Iota with an argument of zero generates an empty vector, this prints as a blank line:

```
      ⍳0

```

There are other ways of generating multidimensional arrays:

```
      A←⍳10
      A
1  2  3  4  5  6  7  8  9  10
      B←A∘.×A
      B
    1    2    3    4    5    6    7    8    9   10
    2    4    6    8   10   12   14   16   18   20
    3    6    9   12   15   18   21   24   27   30
    4    8   12   16   20   24   28   32   36   40
    5   10   15   20   25   30   35   40   45   50
    6   12   18   24   30   36   42   48   54   60
    7   14   21   28   35   42   49   56   63   70
    8   16   24   32   40   48   56   64   72   80
    9   18   27   36   45   54   63   72   81   90
    10   20   30   40   50   60   70   80   90  100
```

This is the so-called outer product: The result element B[I;J] is calculated as A[I]*A[J]. But we can use any scalar operator:

```
      A∘.≥A
 1 0 0 0 0 0 0 0 0 0
 1 1 0 0 0 0 0 0 0 0
 1 1 1 0 0 0 0 0 0 0
 1 1 1 1 0 0 0 0 0 0
 1 1 1 1 1 0 0 0 0 0
 1 1 1 1 1 1 0 0 0 0
 1 1 1 1 1 1 1 0 0 0
 1 1 1 1 1 1 1 1 0 0
 1 1 1 1 1 1 1 1 1 0
 1 1 1 1 1 1 1 1 1 1
```

Another important operator is the reduction, illustrated with an example:

```
      A←3 4⍴⍳12
      A
   1   2   3   4
   5   6   7   8
   9  10  11  12
      +/[1]A
15  18  21  24
      +/[2]A
10  26  42
      +/A
10  26  42
```

+/[1] does a reduction along the first coordinate: It calculates the sum of the rows. +/[2] sums the columns of the matrix. Omitting the index does a reduction along the last coordinate. If you want to add all the elements of A it can be done in two ways:

```
      +/+/A
78
      +/,A
78
```

The monadic comma function (ravel) transform a matrix of any dimension into a vector.

This small example calculates e in two different ways:

```
      )DIGITS 16
WAS 10
      +/÷!¯1+⍳20
2.718281828459045
      ⋆1
2.718281828459045
```

You can use indices to take out parts of a matrix:

```
      A
   1   2   3   4
   5   6   7   8
   9  10  11  12
      A[2;3]
7
      A[2;]
5  6  7  8
      A[;3]
3  7  11
      A[1 2;3 4]
  3  4
  7  8
```

The indices a specified in square brackets and separated by a semicolon. The indices may be a scalar, vector, or matrix in any dimension. Omitting an index means take the whole row or column.

It is also possible to transpose a matrix, using the transpose operator ⍉, this is written as the circle (upper case o) ○, backspace, and backslash \:

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

```
  3   7  11
  4   8  12
```

This also works for higher dimensions:

```
      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
      3 2 1⍉A
  1  13
  5  17
  9  21

  2  14
  6  18
 10  22

  3  15
  7  19
 11  23

  4  16
  8  20
 12  24
      1 3 2⍉A
  1   5   9
  2   6  10
  3   7  11
  4   8  12

 13  17  21
 14  18  22
 15  19  23
 16  20  24
      ⍉A
  1   5   9
  2   6  10
  3   7  11
  4   8  12
```

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

The dyadic transpose should have a permutation of the index numbers as the left argument. The monadic transpose exchanges the last two indices. By repeating the same index number it is possible to extract the diagonal of a matrix:

```
       A←3 3⍴⍳9
       A
   1  2  3
   4  5  6
   7  8  9
       1 1⍉A
1  5  9
```

APL can also manipulate character strings. They are written in quotes. A string with a single character is a scalar, longer strings are vectors:

```
       A←'X'
       B←'A LONGER TEXT'
       ⍴A

       ⍴B
13
       B[3 4 5 6]
LONG
```

Programming

Simple programs are created by writing the del character (∇) followed by the name of the program:

```
       ∇HELLO
[1]    'HELLO WORLD'
[2]    ∇
       HELLO
```

```
HELLO WORLD
```

The machine writes the line numbers automatically.

You can also write operators that return a value:

```
      ∇Z←A PLUS B
[1]   Z←A+B
[2]   ∇
      7 PLUS 9
16
      (7 PLUS 9)×(3 PLUS 4)
112
```

See 3.17pp in the User's Manual on how to edit your program to correct mistakes.

To get a list of all defined functions in your workspace:

```
      )FNS
HELLO   PLUS
```

And a list of all variables:

```
      )VARS
A
```

Your workspace is volatile, it will be lost when you log out. So remember to save it before shutting down:

```
      )SAVE TEST1
  14.12.02 03/07/14
```

You can easily replace the current workspace with a saved one:

```
      )LOAD TEST1
SAVED  14.12.02 03/07/14
```

Get a list of your saved workspaces:

```
      )LIB
```

```
COVER
SUDOKU
TEST1
```