A Practical Introduction to

APL 1 & APL 2

by

Graeme Donald Robertson

. . TRAINING THAT WORKS . . .

Date: ....

Place: ....

Instructor: ....

Student(s): ....

....

....

....

....

....

ROBERTSON (Publishing)
15 Little Basing, Old Basing,
Basingstoke, RG24 8AX, UK.

First edition published March 2004 as APL1&2.PDF
Second edition published September 2004 as APL1_2.PDF
Third edition published January 2008 as APL1&2.PDF

ISBN 0 9524167 1 9

Conduct of this 2 day course:

After short introductions, the student group is invited to divide up into pairs.
Each pair works on one computer/terminal for the duration of the course.
Each pair is given the first lesson and asked to work through it on their computer at their own pace.
Pairs are encouraged to help each other with new concepts and difficulties as they arise and to experiment on the computer with any ideas which they think they can express in APL statements.
Tuition is given when problems cannot be resolved by the pair. Questions may be answered directly on matters of fact, or indirectly by way of a suggestion as to how the problem might be tackled.
Each day covers about 7 lessons, depending upon the pace of each pair.
There is no pressure to complete all lessons (remaining notes are given out at the end of the course).
At the discretion of the tutor, lessons may be skipped or assigned for private study after the course has ended.
Short synopses are given (with an overhead projector or white board) at suitable intervals throughout the course to the group as a whole.

A Practical Introduction to

APL 1&2

APL is the only language to have been 200 years in the debugging.
Ken Iverson

Day 1: First Generation – APL 1

APL 1 - Core APL

Session 1

- APL Character Set
- APL Keyboard
- Primitive Functions
- Simple Arrays
- Assignment of Variables
- Indexing & Special Syntax
- Error Messages

Session 2

- User-Defined Functions
- Editing Functions
- Local & Global Variables
- Order of Execution of Functions
- System Commands
- APL Idioms

...learning by practice, by induction, and by heuristic methods...
...pragmatic teaching by encouraging experiment and by individual tutorial.
Ken Iverson

L E S S O N 0

Why Learn APL?

APL is a high-level, general-purpose, *intuitive programming language* which is designed to be easy on the programmer even if consequently hard on the computer - through *power*, not inefficiency.

APL has its own special character set of around 200 alphabetic characters and symbols. Although the APL symbols might appear illegible and unintelligible, each individual symbol performs a specific task making programs very concise. APL is A Programming Language which is essentially *simple and easy to learn*, and APL is interactive making it possible to *experiment with different ideas* while developing solutions.

Getting to Know Your APL Keyboard

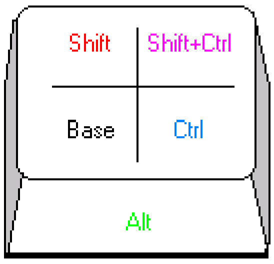
Your computer should be set up already so that an APL session is visible and has the focus. Typing on your keyboard should cause characters to be displayed on your screen. Try typing something. When you come across a new symbol, or key combination, write it on the supplied blank keyboard. This will help you quickly to become familiar with the new APL key layout.

Symbols which require the *Shift* key to be pressed should be written in the upper left hand corner of the corresponding key cap on the supplied blank keyboard chart. Symbols which require the *Ctrl and Shift* keys to be pressed should be written in the upper right hand corner of the corresponding key on the chart. Symbols which require the *Alt* key to be pressed should be written on the front of the corresponding key, as shown below. (Beware of *Ctrl* keys on a mainframe.)

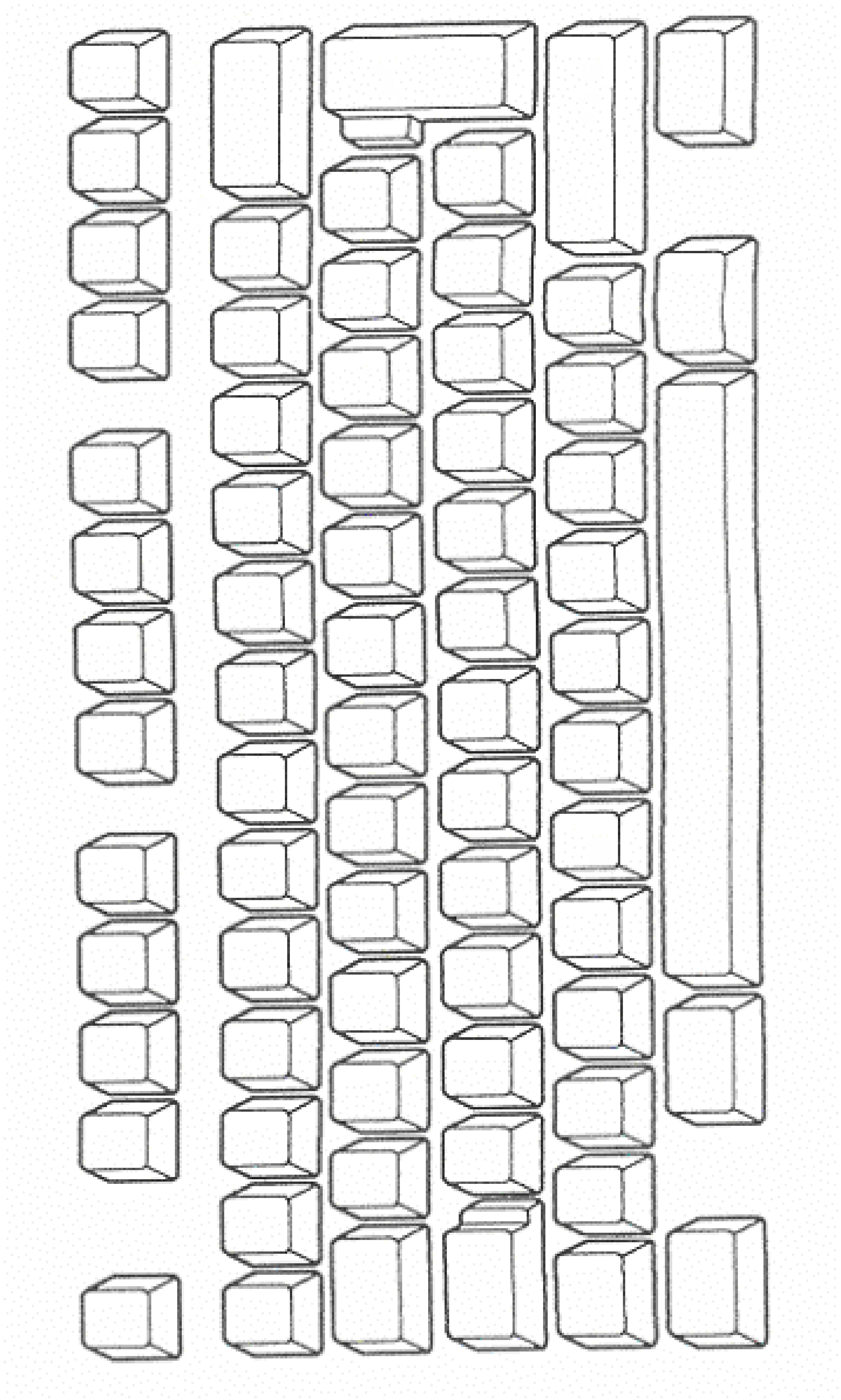
- Type the numbers 0 to 9 on your keyboard and write them in the lower left hand corner of the corresponding key on your keyboard chart. Type in the upper case letters A to Z on your keyboard and write them in the appropriate positions on the keyboard chart.

- Find the symbols + and − on the keyboard and write them on your chart. Find the symbols × and ÷ on the keyboard and write them on your chart. Use the backspace key to rub out the typing. Put that key on your chart too.

- In APL each expression which is typed into the APL session is executed when the Enter key is pressed. In mainframe APL2 the key which is used to enter expressions is the right Ctrl key, and possibly also the numeric pad Enter key. Mark the appropriate key on your chart.



- Ask your tutor for LESSON 1.



LESSON 1

Simple Arithmetic Expressions

Go at your own pace. Experiment. Try to work it out. Think. Talk about it.

- Use APL to add any two numbers together. Check the result. For example, type

```
      65.35 + 35.65
101
```

*Hint: Hit the **Enter** key when you are ready to execute the line containing the cursor.*

Notice how, in *immediate execution mode*, APL indents the cursor 6 spaces to indicate that it is ready to accept the next line of user input. Everything which has been input by the user is indented by 6 spaces, and is **coloured green** in mainframe APL2. Output from the computer starts at the left hand margin and is **coloured red**, as are error lines.

- Type the following two lines into your session and explain the results.

```
      14 - 9
5
      - 7
¯7
```

Notice the distinction between the **negate** function (-) and the **negative** sign**,** or high minus symbol, (¯), which is an intrinsic part of a number, like the decimal point.

Symbols such as - and + can be used either with a *right argument* (which is called the *monadic* or prefix case) or with a *left and right argument* (which is called the *dyadic* or infix case). Thus the hyphen symbol can be used monadically to mean negate or dyadically to mean **minus,** or subtract. Write the new high minus symbol on your keyboard chart.

The **plus** and **minus** signs were introduced by the German mathematician Johann Widmann in 1489 to signify addition and subtraction. Dyadic deployment of the symbols is now familiar to everyone.

APL has many such powerful primitive functions which allow complex computations to be done very easily. Primitive functions follow the principle of one symbol per mathematical operation.

- Experiment to see if you can deduce the monadic and dyadic meanings of the symbols × ÷ | ⌈ ⌊ * ⍟ ! both by applying simple numeric arguments, and by inference from the form of the symbol itself.

The **times** sign was introduced by the English mathematician who invented the slide rule; William Oughtred (1575 - 1660). Its use to signify multiplication is now familiar to everyone who has been exposed to the language of elementary algebra. Most computer languages use * to indicate multiplication. Algebraists use a variety of alternative ways to indicate multiplication: a×b or a.b or ab. APL consistently uses ×. "APL is the only [computer] language to have been two hundred years in the debugging," says Iverson.

APL is derived from mathematical notation. It did not appear from the standard evolutionary origins of most other computer languages. APL crystallized from an unconstrained theoretical notation (*Iverson notation*) when it was realized that it could be executed on a computer. "I wasn't trying to design or implement a language for a machine," confessed Iverson.

The monadic meanings of `× ÷ | ⌈ ⌊ * ⍟` and `!` are **direction**, **reciprocal**, **magnitude**, **ceiling**, **floor**, **exponential**, **natural logarithm** and **factorial** respectively, and dyadic meanings **multiply**, **divide**, **residue**, **maximum**, **minimum**, **power**, **logarithm** and **binomial** respectively.

- Investigate the monadic meaning of `+` Experiment with any suggestive arguments! ;-)

The *result* of one expression can be used as the argument to another function.

- Try some compound expressions such as

```
      3 × 4 + 6
30
```

and

```
      (3 × 4) + 6
18
```

Hence explain the result of the expression

```
      14 - 6 - 5 - 3 - 7
17
```

Beyond BIDMAS. Remember BIDMAS (or BODMAS)? It tells you the order of precedence in simple arithmetic expressions – brackets first, then indices (or of), division, multiplication, addition and finally subtraction. APL, on the other hand, does not assume any special order of precedence between functions. Execution simply proceeds from right to left unless you use **parentheses** (round brackets) to control the order of execution. All APL functions have equal priority. This basic “right-to-left” grammatical rule applies to dyadic functions and monadic functions alike in APL. (It’s a bit like the rule in English that the object of a sentence comes after the verb.)

Rule 1: The right argument of any function, monadic or dyadic, is the result of the entire expression immediately to its right.

Some functions take boolean arguments and return boolean results.

- Reading 1 as true and 0 as false, verify the truth values of these expressions.

```
      1 ∨ 1
1
      1 ∧ (0 ∧ 1) ∨ 1 ∨ 0
1
      ~ 0 ∨ ~ 0
0
```

These invoke the simple logical functions: **and** (∧), **or** (∨) and **not** (~).

- Some functions take numeric arguments and return boolean results. Verify the results of

```
      20.5 = 41 ÷ 2
1
      101 < 200 - 100
0
      27.3 > 39.31
0
```

These introduce binary relational functions: **equals** (=), **less-than** (<) and **greater-than** (>).

- Trigonometric functions are implemented via the dyadic **circle** function. A left argument of 1 returns the sine of the right argument. Assess the result of

```
      1 ○ 3.14159
0.000002654
```

knowing that Sin(π) is zero. A left argument of 2 returns the cosine, 3 returns the tangent.

The left argument of the circle function may be an integer between 12 and -12 representing various standard pythagorean, trigonometric, hyperbolic and complex number functions.

- Explore a few examples. Find the meaning of the monadic circle function.

- Type the following line into your session and execute it a few times.

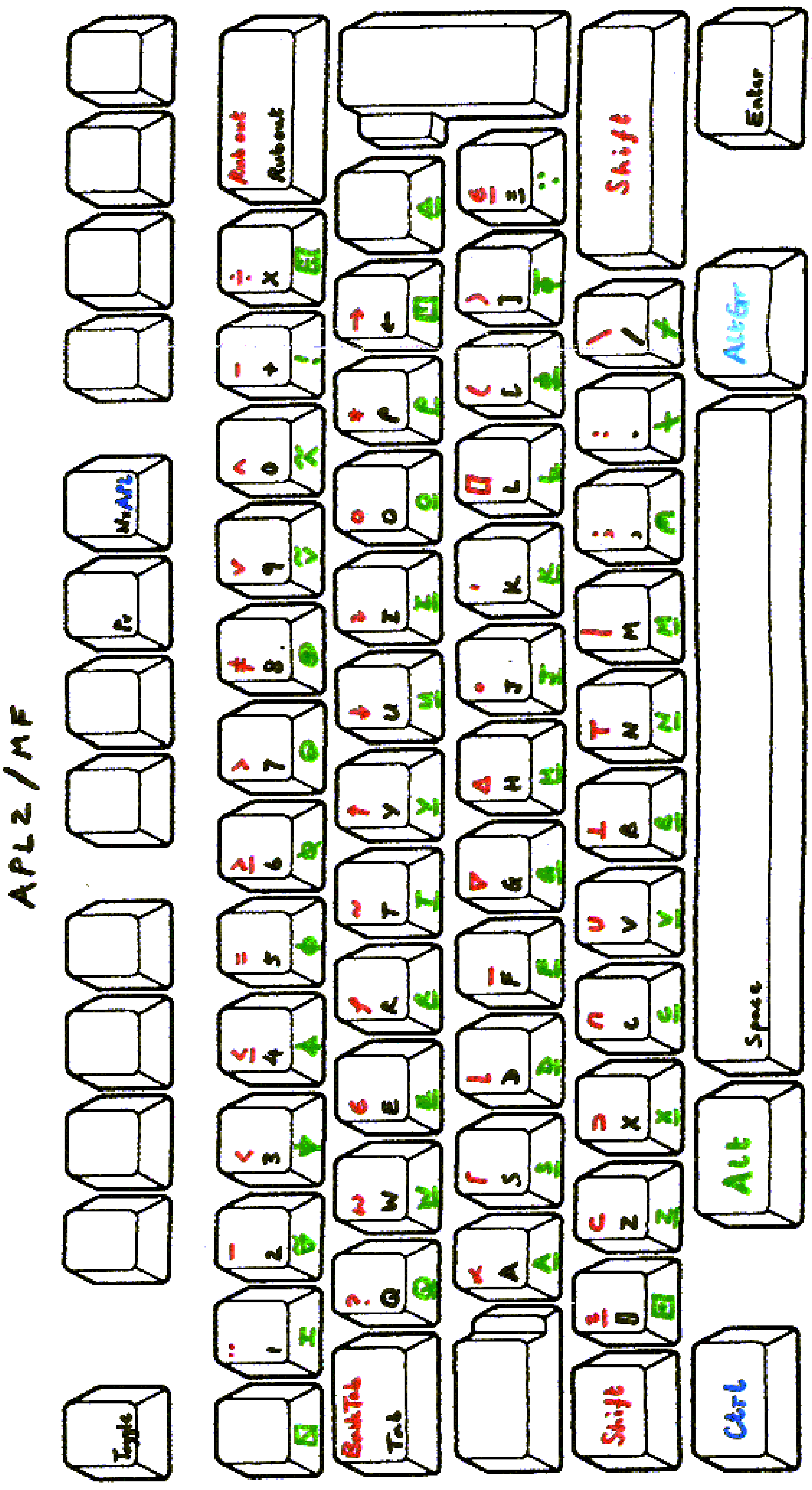
```
      ? 6
5
```

What do you think the results indicate about the meaning of roll (?)? ☺

- Try some more adventurous examples of the application of Rule 1.

```
      ⌊0.5+3.23
3
      ⌈¯0.5+3.23
3
      ⌊¯3.5⌈¯8.2
4
      3.2|¯4.3
2.1
      ((?5)≤?8)∨(?5)≥?8
1
      (~1⍲0⍱1)=1≤0.2≤3≤4
0
      (2!6)=(!6)÷(!2)×!4
1
      ((3×4)=*(⍟3)+⍟4)∧(3÷4)=*(⍟3)-⍟4
1
```

- Ask your tutor for LESSON 2.


APL2 / MF
Toggle
Rub out
Rubout
Back Tab
Tab
Shift
Enter
Alt Gr
Space
Alt
Ctrl

LESSON 2

Names, Lists & Literals

Numbers and results of expressions can be assigned to names. The left assignment arrow (←) may be read as **gets** or **is-assigned** or simply **is**.

- Enter the following statements (or sentences), noting that ⋆ means power and × means times,

```
      INTEREST ← 0.09
      YEARS ← 6
      VALUE←500×(1+INTEREST)⋆YEARS
```

- Type in the name of a variable and hit the Enter key to display the contents of the variable in the session.

```
      VALUE
838.6
```

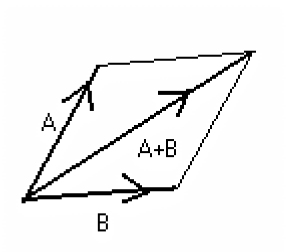
Generalized Scalar Functions. Some functions that take single numbers (**scalar**) arguments have well defined behaviour when the arguments are extended to lists of numbers (**vector**).

- Execute the line

```
      1 2 3 + 4 6 8
5 8 11
```

and explain the result.

Addition of vectors is not a new concept. Newtonian mechanics (c. 1687) employs 3 element vectors to describe positions, velocities and forces in 3D space. Addition of forces may be represented
by lines and parallelograms, as below. The sum is calculated by *vector addition*, using the **plus** sign, as above.



N-dimensional vector spaces (containing N-element vectors) are now employed routinely in many branches of pure and applied mathematics. Indeed the concept of a vector space is one of the principal unifying concepts in the whole of mathematics (see Hilbert Space in Wikipedia).

"The use of a programming language in which elementary operations are extended systematically to arrays provides a wealth of useful identities," says Dr Kenneth Iverson in his book *A Programming Language, Wiley 1962.*

APL adopts this element-by-element approach to vector addition and generalises it to many standard mathematical functions, taking dyadic **plus** and monadic **negate** as role models, or templates.

- Check the results of

```
      - 3 4 5
¯3 ¯4 ¯5
      - 3 4 ¯5 4 6 5.0 ¯8.567
¯3 ¯4 5 ¯4 ¯6 ¯5 8.567
      1 2 3 × 2 ¯2 2
2 ¯4 6
      45 ¯3 2 2.33 + 99 7 4 0.4
144 4 6 2.73
```

- Explore other expressions, using lists of numbers as arguments to the primitive scalar functions represented by symbols `+ - × ÷ | ⌈ ⌊ * ⍟ ○ !`

Scalar Extension. If one of the arguments of a scalar dyadic function is a scalar and the other is a vector (or list) then the scalar is automatically extended to have the same length as the vector.

- Enter

```
      1 ○ .1 .2 .3
0.09983 0.1987 0.2955
```

- Compare with

```
      1 1 1 ○ .1 .2 .3
0.09983 0.1987 0.2955
```

and

```
      1 2 3 ○ .1 .2 .3
0.09983 0.9801 0.3093
```

Otherwise, if the arguments have incompatible lengths then a `LENGTH ERROR` is reported.

- Try to execute the following line.

```
      1 2 ○ .1 .2 .3
```

Literals. Variables can be assigned to lists of literal characters as well as to lists of numbers. Character strings have to be enclosed inside APL *quotes* in order to distinguish literal characters from defined names or simple numerics.

- Enter your name and web address, e.g.

```
      NAME←'DEBBIE ROBERTSON'
      ADDRESS←'APL4.NET'
```

The dyadic structural functions **catenate** (,) **take** (↑) and **drop** (↓), and the monadic structural function **reverse** (⌽), can be used on any list of numbers or characters to produce a new related list.

- Try

```
      ⌽NAME
NOSTREBOR EIBBED
      7↑NAME
DEBBIE
      E←(6↑NAME),'@',ADDRESS
      E
DEBBIE@APL4.NET
```

- Monadic use of Greek letter rho (function **shape**) returns the number of elements in the vector `NAME`. Check the result of

```
      ⍴NAME
16
```

- Dyadic rho (**reshape**) returns the right argument reshaped to have exactly the number of elements specified by the left argument. Try

```
      4⍴ADDRESS
APL4
      40⍴ADDRESS
APL4.NETAPL4.NETAPL4.NETAPL4.NETAPL4.NET
      ⌽50⍴NAME,' '
NOSTREBOR EIBBED NOSTREBOR EIBBED NOSTREBOR EIBBED
```

The shape of a vector is the number of elements in the list.

- Type

```
      ⍴NUMS←56 87 75 80 79 86 84 90
8
      ⍴CHARS←'56 87 75 80 79 86 84 90'
23
```

Literal digits can be converted into numbers using the very powerful **execute** (⍎) function (which is said to make APL 'self-conscious') and numbers can be converted into characters using the very useful **format** (⍕) function.

- Explain the results of

```
      ⍴⍕NUMS
23
      CHARS=⍕NUMS
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
      ρ⍎CHARS
8
      NUMS=⍎CHARS
1 1 1 1 1 1 1 1
      CHARS,⍕NUMS
56 87 75 80 79 86 84 9056 87 75 80 79 86 84 90

      ⍎(3×5)ρ'NUMS⋄'
56 87 75 80 79 86 84 90
56 87 75 80 79 86 84 90
56 87 75 80 79 86 84 90
```

The *diamond* symbol (⋄) is not a function. It is a statement separator. You might not be able to find it on your APL2 mainframe keyboard. However, diamond is an example of an overstruck character – from the days when space for characters was scarce. A diamond can be input using the three consecutive symbols <_> where _ is the printable backspace. This requires that you first type the command

```
      )PBS ON
```

in APL2, or switch to replace mode via the Insert key in Dyalog APL.

Interval (⍳) can be used to generate any uniformly spaced range of numbers.

The monadic meaning of the iota character (⍳) is a function called **interval** or index generator. It takes a scalar argument and returns a vector result.

- Try

```
      ⍳9
1 2 3 4 5 6 7 8 9
      ⍳19
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
      ¯1+(⍳19)÷10
¯0.9 ¯0.8 ¯0.7 ¯0.6 ¯0.5 ¯0.4 ¯0.3 ¯0.2 ¯0.1 0 0.1 0.2 0.3 0.4 0.
5
      0.6 0.7 0.8 0.9
```

- Experiment with examples like

```
       ¯0.3×8-⍳15
¯2.1 ¯1.8 ¯1.5 ¯1.2 ¯0.9 ¯0.6 ¯0.3 0 0.3 0.6 0.9 1.2 1.5 1.8 2.1
```

- Now try

```
      ¯1+(⍳99)÷50
```

or

```
      ¯1+(⍳9999)÷5000
```

- Use the appropriate keystroke (usually *Ctrl+C* on a mainframe keyboard or *Ctrl+Break* on a PC) to interrupt execution of lengthy or verbose operations. Write this important key combination on your keyboard chart. Learn to interrupt without compunction. Waiting for a rogue function to finish may be *very* expensive on a mainframe. You control the computer now.

APL primitive functions appear *atomic* in the sense that they never stop half way through. They either finish completely or appear not to have started at all. Therefore breaking an APL process always leaves the processing stack at a definite given point in an APL program.

APL *idioms* are commonly used combinations of tokens. They are phrases that are immediately recognised by APL programmers when reading APL code. A simple example of an idiom is

```
      ⍳⍴NUMS
1 2 3 4 5 6 7 8
```

which returns a count of the elements in the vector *NUMS*.

- Propose a use for this idiom:

```
      (1↓NUMS)-(¯1↓NUMS)
31 ¯12 5 ¯1 7 ¯2 6
```

Note the occasional judicious use of redundant parentheses to enhance readability.

- Experiment with dyadic functions ⍴ ↑ and ↓ using *scalar* (single) integer left arguments and *vector* (list) numeric (or character) right arguments. Note, in particular, the **shape** of the arguments and the shape of the results.

- Write an expression which rounds a number of pennies to the nearest 12p. Andrew, James, Charles and Marcus each have a building society account: these contain £5,081.09, £11,954.55, £812.97 and £6,241.00 respectively. Each account has a different annual interest rate: 4.1%, 3.5%, 2.6% and 3.25%. Write an expression which returns the interest on each account. Write another expression which returns how much each person could have at the end of ten years of saving, to the nearest 12p?

LESSON 3

Indexing Non-Scalar Arrays

In a simple and intuitive manner, square **brackets** [ ] are used to select items from a list.

- Enter

```
      NUMS[3 2]
75 87
      (⌽⍳99)[NUMS]
44 13 25 20 21 14 16 10
      (33 44 55 66)[3 2 3 3 1]
55 44 55 55 33
      'scarlet'[1 6 2 4 6 7]
secret
```

The **shape** of the result is the shape of the **index**. If no index is included within the brackets (elided index) then the whole vector is returned. e.g.

```
      NUMS[]
87 75 80 79 86 84 90
```

- Use bracket indexing to select the smallest and largest elements from the vector

```
      A ← ? 100 ⍴ 1000
```

*Hint: Monadic **grade-up** (⍋), applied to argument A returns the permutation vector which would sort A in ascending order.*

Matrices. We have generalised the arguments of functions from *scalars* to *vectors*, or lists. Now we generalise further to *matrices*, or tables.

When the concept of a matrix of numbers is first encountered in mathematics it can appear quite forbidding, but they have many uses. For example, they form the bases of representations of continuous groups which have many deep applications in science. We here consider a matrix simply as a rectangular table of numbers or characters.

In order to create a vector we may use the dyadic **reshape** function (⍴), with a *single* numeric left argument, to produce a list of that length containing elements taken consecutively from the right argument. In order to create a matrix we may use the reshape function with a *two element* numeric vector left argument to produce a table which has that number of rows and columns.

- Examine the displayed output from

```
      3 4 ⍴ 999
999 999 999 999
999 999 999 999
999 999 999 999
```

```
      5 5 ⍴ ⍳ 25
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
      2 13 ⍴ 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
ABCDEFGHIJKLM
NOPQRSTUVWXYZ
```

- Create a character matrix called MONTHS that has 12 rows and 9 columns formed from the carefully spaced character string

```
      'JANUARY  FEBRUARY MARCH    APRIL .. DECEMBER '
```

- Use bracket indexing to select rows 3 and 4 and columns 1 and 2.

```
      MONTHS[ 3 4 ; 1 2 ]
MA
AP
```

Note the semicolon to separate dimensions.

In APL, there are often (always?) many ways to accomplish the same task (although some solutions are manifestly more elegant than others).

- Use take and/or drop with 2 element left arguments to produce from `MONTHS` the selection:

```
      MONTHS[ 3 4 9 ; 5 1 2 ]
HMA
LAP
ESE
```

- From the result of

```
    (12↑0 0 1)⌿MONTHS
MARCH
```

deduce the meaning of **compress-first** (⌿) with boolean left argument. How could you use this function and **compress** (/) to make the above selections? Why was compress renamed **replicate** when the left argument was generalized to include integers rather than just booleans?

*Suggestion: Look up **replicate** (/) in your reference manual or help file.*

- Create a variable called `SALES` which has 12 rows and 3 columns and is filled with random numbers between 1 and 1000.

- Select the first row of `SALES` (and all the columns - by eliding the columns entry).

```
      SALES[1;]
935 384 520
```

- Select the third and first column (and all the rows - by eliding the rows entry).

```
      SALES[;3 1]
520 935
 54 831
  8 530
    .
    .
    .
```

Structure functions. The primitive selection functions, which were used for manipulating lists, all generalise to matrices. We have already seen **take** (↑) and **drop** (↓) applied to matrices. There are other primitive functions that change the structure of their arguments.

- Experiment with monadic transpose (⍉) on SALES and other matrices.

```
      ⍉SALES
935 831 530 384 687 847 654 911  48 633 366 723
384  35 672  67 589 527 416 763 737 757 248 754
520  54   8 418 931  92 702 263 329 992 983 652
```

- Experiment with **reverse** (⌽) on SALES and other vectors and matrices.

```
      ⌽SALES
      ⍉⌽⍉SALES
```

- Look up **take**, **drop** and **reverse-first** (⊖) in your reference manual or help file. Become familiar with your sources of reference.

- Make a report.

```
      REPORT←MONTHS,⍕SALES
```

- Explain the **expand** function (\) and use it to double the width of the report.

```
      ((2×¯1↑⍴ REPORT)⍴1 0)\REPORT
J A N U A R Y       9 3 5   3 8 4   5 2 0
F E B R U A R Y     8 3 1     3 5     5 4
M A R C H           5 3 0   6 7 2       8
A P R I L           3 8 4     6 7   4 1 8
M A Y               6 8 7   5 8 9   9 3 1
J U N E             8 4 7   5 2 7     9 2
J U L Y             6 5 4   4 1 6   7 0 2
A U G U S T         9 1 1   7 6 3   2 6 3
S E P T E M B E R     4 8   7 3 7   3 2 9
O C T O B E R       6 3 3   7 5 7   9 9 2
N O V E M B E R     3 6 6   2 4 8   9 8 3
D E C E M B E R     7 2 3   7 5 4   6 5 2
```