A
2₀ - Day
Introduction to

APL 3 & 4

Graeme Robertson Ltd.

Featuring DYALOG Versions 10 & 11

Day 1: Third Generation APL – Objects

Modules 1 to 10

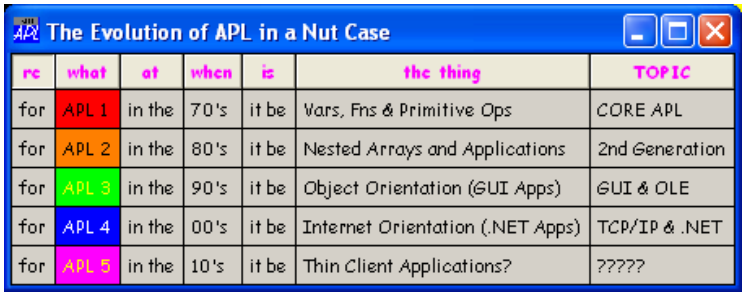
- Graphical User Interface
- Object Linking and Embedding

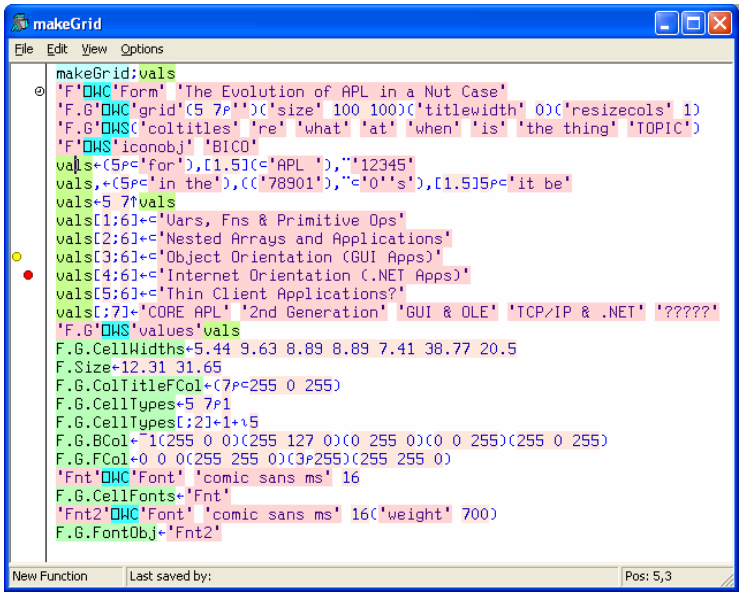
Day 2: Fourth Generation APL – The Internet

Modules 11 to 20

- Transmission Control Protocol / Internet Protocol
- Microsoft .NET Framework

Learning Modules





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

Published January 2007 as APL3_4.PDF.*

Dyalog APL is worth learning because it:

- strictly complies with APL ISO **Standard** 8485
- has a brilliant **tracer** for the interpreted APL environment
- runs on **Windows** (9x, NT, 2000, CE, XP), AIX, Solaris, Linux and more…
- affords **multilingual** development and runtime environments
- has 71 built-in **GUI classes** (based on Windows API), each with 10 to 80 properties…
- is **.NET enabled**, ready for Windows Vista (the 2007 successor of XP)
- has excellent printed **manuals** and detailed **help** files (created using APL)
- has many useful supporting sample **workspaces** and files
- provides DSS, the *Dyalog Support Service* via www.dyalog.com and support@dyalog.com

A Level 1 course in **APL1_2.PDF** introduces mainstream 1st and 2nd generation APLs such as IBM APL2, Dyalog APL, STSC APL*PLUS and MicroAPL APL.68000. **APL1_2.PDF** is freely available from MicroAPL at http://www.microapl.co.uk/apl/APL1_2.PDF and may be given by instruction.

This ***Level 2*** course in **APL3_4*.PDF** introduces Dyalog APL versions 7-11 under Microsoft XP Pro. This course, when delivered by Graeme Robertson Ltd., comes bundled with supporting materials:

- **varChar**, a Dyalog APL addin for studying Dyalog APL arrays,
  (a free stand-alone version is available from www.dyalog.com download zone)
- a handy, short, fold-up Dyalog APL **Reference Card**
- a sample Pocket Dyalog application workspace called **AddrBk.DWS**

Conduct of this course: The suggested conduct of the course is similar to that stated in **APL1_2.PDF** and reproduced below. The course notes are provided in Portable Document Format, **one APL3_4Module$_n$.PDF file at a time**, to enable cut & paste, to encourage experimentation and remote learning, and to increase the likelihood of obtaining feedback ☺.

> After short introductions, the group is invited to divide up into pairs. Each pair works on one computer for the duration of the course. Each pair is given the first module 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 that they think they can express in APL statements. Tuition is given when the pair cannot resolve problems. 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 could cover about 10 modules, depending upon the pace of each pair. There is no pressure to complete all modules (remaining modules are given out at the end of the course). At the discretion of the tutor, modules may be skipped or assigned for private study after the course. Introductions and synopses are given with an overhead projector at suitable intervals throughout the course to the group as a whole.

APL 3 & 4
Course Contents

Some innocent facts ☺

- There is no exact finite Boolean representation of the decimal number 0.1 as a binary floating-point number.
- The density of rational numbers on the real line is infinitesimal compared to the density of transcendentals.
- One Jupiter day equals 0.416.. Earth days.

Question ☻ Is there a planet somewhere whose day is approximately equal to 10 Earth days?
Answer ☻ Very likely! (On that planet, this is a two day teach-yourself course ☺.)

Module0: Notation and Conventions

§ 0.1 New Symbols

It shouldn't worry an APLer to have a few new suggestive symbols kicking around. Of course they ought to be well defined, although their meaning should be 'onomatopoeically' self-evident from their shape. Most of our new symbols are not to be found in the ⎕AV of any APL font and are simply introduced to clarify explanation and writing in the spirit of the original Iverson Notation. Some of these symbols are to be found in some APL fonts, but they are not here coloured green and are, by implication, not available as part of an executable Dyalog APL statement. Executable APL keywords are written in green.
An executable input expression is written in green APL font and indented by 6 spaces. Any resulting output from the expression is coloured red and is placed unindentedly on the subsequent line or lines. Or, it is placed on the same line as the input expression, separated from it by the new symbol ↳ .

`Expr ↳ Result`	Error unless expression `Expr` returns result `Result`

The symbol ↳ (*returns*) is black to identify it as non-executable, and the definition is in black to emphasise that is not genuine (current) APL. The `Result` may also be an executable expression which itself would, if executed, return a result identical to that returned by the original expression, `Expr`. So, a valid example would be `3×4 ↳ +/3⍴4`

`Expr ↦ Obj`	`Expr` instantiates object `Obj` (as a side effect)

This is used to comment functions that internally create global GUI objects.

`Expr ↪ Function`	`Expr` can fire callback function `Function`

This is used to document functions that internally assign callbacks to global objects.

`APLCode_ ↵`	Code continues on next line …
`_APLCode ¶`	.. end of line of code

The pair (↵,¶) is used to wrap 2 or more input lines while maintaining their syntactic unity and integrity.

`⊢Prop`	Error unless proposition `Prop` is contingently true

Symbol ⊢ (*contingently*) means factually true, so here `Prop ↳ 1`. `⊢a∨b` implies, *a posteriori*, that `a∨b↳1`

`⊨Prop`	Proposition `Prop` is necessarily true, so `Prop ↳ 1`

Symbol ⊨ (*logically*) establishes the logical truth of a proposition, so `⊨a∨b` implies, *a priori*, that `a∨b↳1`, and further, that (`⊨a`) *or* (`⊨b`), where *or* is inclusive rather than exclusive.

`Code .. Code`	Means fill in the gap with the obvious code

This single character double-dot (*gap*) can also be used at the very start of a line to mean fill in the obvious start.
Note in the modules to follow, boxed syntax definitions usually just apply to one of a number of possible syntaxes. For example, ambivalence and shy results are not usually left undetermined.

`Code …`	Means finish the statement with the obvious ending

This single triple-dot character (*ending*) asks you, the reader, to complete the extreme end of any expression or suitably parenthesised expression. It asks more of you than *dittos* which just imply repetition, and more than gaps (..) which are usually essentially repetition too, given the clues from the code on either side. An author who uses this ending should feel that the code to the left is sufficient for the reader to be able to fill in the ending without undue pain. Suitable fill code should spring to mind.

You may consider .. or … as notation for *x* or *y* in some 'literal algebra'. *eg* "Hello .., this is .., how are you … ?" where .. and … are any suitable strings.

```
      ⍬                                    ⊨⍬≡⍬
```

This implies that `⍬≡⍬↳1` and that `⊨⍴⍬≡⍴⍬`

Other symbols that may be introduced in the following modules include:
f_1 as a generic monadic function (distinguish *monadic* function from *monistic* operator),
g_2 as a generic dyadic function (distinguish *dyadic* function from *dualistic* operator),
$\hat{o}_a$ to represent a typical operator and
$\tilde{n}_a$ to represent a typical namespace.

§ 0.2 Naming Conventions

The following naming conventions have been followed where possible and where appropriate:

Variable names are regarded as proper nouns and therefore start with a capital letter, an exception being local *temporary variables* used near to their source. These may be regarded as pronouns for which we may use small letters.

Functions and operators play the roles of verbs, adverbs, adjectives, conjunctions or prepositions depending on the details of their syntax. Their names all start with a small letter as they are mid-sentence words. However, *niladic functions* may start with a capital, because if they return a result they are syntactically like variables, and if they do not return a result they are like complete imperative intransitive verb sentences. Likewise, *monadic functions that do not return a result* may also begin with a capital letter because they are like imperative transitive verbs and therefore usually begin a sentence.

To give you the idea, an attempt has been made to compose the Course Contents roughly in this style. Module titles are noun phrases that represent the subject matter. Section and subsection headings are subjects or predicates or object noun phrases. An attempt is made to capitalize the words according to their contextual parsing character (see APL Linguistics in *Vector Vol.2 No.2 p118*).

Sentence Syntactic Elements	**Grammatical Rôle**	**Example Names**
`Variable`	Noun/Pronoun	`Data ⋄ DATA ⋄ d`
`Niladic`	Intransitive imperative	`Run ⋄ RunApplication`
`R←Niladic`	Noun	`.. is Temperature`
`Monadic ⍵`	Transitive imperative	`RunOn ...`
`R←monadic ⍵`	Adjective/Participle	`.. is dataType ...`
`⍺ dyadicFn ⍵`	Verb	`.. takes ...`
`R←⍺ dyadic ⍵`	Preposition/Conjunction	`.. is .. tiedTo ...`
`R←(⍺⍺₁ monistMonadOpr)⍵`	Adverb	`.. completely ...`
`R←⍺ (⍺⍺₁ monistDyadOpr) ⍵`	Adverb	`.. overNight ...`
`R←(⍺⍺₁ dualMonadOpr ⍵⍵₁) ⍵`	Conjunction/Preposition	`.. togetherWith ...`
`R←⍺ (⍺⍺₁ dualDyadOpr ⍵⍵₁) ⍵`	Conjunction/Preposition	`.. is .. and ...`
etc...		

This is a heuristic pragmatic association with ordinary language for the purpose of meaningful readability. Note the convention of capitalizing the first letter of each word or syllable after the first.

Object names follow variable naming conventions. Object method names should follow function conventions but are predetermined by their author. From a glance at IE6 method names (or OpenGL method names) Microsoft may be moving in the above direction. Object property names are nouns and are generally 'correctly' (by this convention) predetermined.

An effort has been made to (capitalize and colour) keywords in the text when used as keywords rather than as general concepts. *eg* an object initiates an event, *cf* an object has an `Event` property. Don't be too reluctant to choose long meaningful names. AutoComplete springs into action in version 10 to ease the pain.

Pre-empting a Unicode (see http://www.unicode.org) `⎕AV`, letters in names assume all the characteristics of modern fonts, *ie* you can have A *A* **A** ***A*** A *A* ***A*** ~~A~~ etc… all for the price of `⎕AV[66]` and all equivalent, just as an ordinary word is assumed to have the same meaning in any colour. The three alphabets in `⎕AV` are taken to be `A..Z` $_{a..z}$ $^{a..z}$ *ie* capital letters, small letters (possibly subscripted) and superscripted letters of various styles. This is done to encourage full tensor notation such as (X_1, X_2, X_3) or

$$T_{ijkl} \leftarrow G_{im} +.\times T^{m}{}_{jkl}$$

(as used in *eg* http://arxiv.org/ftp/hep-th/papers/0304/0304244.pdf) and also to encourage traditional British shorthand forms such as W^{m} Robertson & C^{o} L^{td} or Sci^{ntfc} & Med^{cl} Net^{wk}. We also suppose, in these notes, that *any* linear Rich Text may follow a comment symbol in the APL Session and we assume that any number of new symbols will one day be available to APL programmers for names.

Menus and other one-click options relating to the application under discussion are placed in square brackets. Thus [File][Load] and [Tools][Options] refer in an obvious way to Session menus.

§ 0.3 Variable *dataTypes*

Different environments use different notations to specify the number of bytes being used and the interpretation of the bits. For example, the **C** programming language, which we meet in `⎕NA` in Module 9, describes numbers as integer, double, long double, float, .. whereas `⎕NA` itself uses notation similar to that used in the Larg (left argument) of `⎕FMT` which itself originates from FORTRAN. More modern definitions of numeric "dataTypes" are written VT_I2, VT_I4, VT_DECIMAL, VT_R4, VT_R8, …

APL, thank goodness, does not generally demand that the programmer be aware in advance of exactly what type of binary representation is being used for any particular piece of data. Mathematica, J and some other modern computer languages are even more forgiving - numbers may even be infinite!

In pure mathematics, occasionally the domain of numbers in which one is thinking has to be generalised to the next set. Whole numbers are a subset of natural numbers which are a subset of the group of integers which is a subgroup of the group of rational numbers which is a subgroup of the field of real numbers which is a subfield of the complex field which is a subfield of the non-commutative quaternion ring which is a subring of the non-commutative and non-associative octonion ring.

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C} \subset \mathbb{H} \subset \mathbb{O}$$

There is a natural hierarchy that is embraced at each level, as and when required, when learning mathematics. Thus when learning, at an early age, that division of integers can land you in the rationals, you are not generally obliged thenceforth to predetermine whether the result of any particular division will result in a number which is integer or one which is non-integer (but necessarily rational).

In APL the actual representation employed for storing a number can change from line to line and from function call to function call without the programmer having to know. The interpreter itself attempts to

discover the most efficient storage representation. However, when communicating through APL with the 'real' computer world beyond APL 1 & 2 you have to start thinking about the machine a bit more. You sometimes have to tell functions in advance what sort of arguments to expect and what sort of results to generate. This is the function *dataType* signature. It helps hardware to know how to store stuff.

Loosely Correlated *dataType* Definitions (internal memory formats)

APL if you had to say…	**⎕*NA***	**C**	**Interface Definition Language (OLE/COM)**	**Visual Basic**	**C#**	**.NET Data Types**
CharSc	C1, T,	`uchar`		Byte	byte, char	System.Byte, System.Char
CharVec	C, T[]	`char[]` `char*`	VT_BSTR, VT_PTR TO VT_BSTR	String	string	System.String
VecCharVec	{T[4]}[5]	`(char*)[]`	VT_ARRAY OF VT_BSTR, VT_STRING[]	String(0 to 12)	string[]	
NumSc, BoolSc, IntSc	I2, I4, F8, U1, U4	`short, uint, int, long, float, dword, lptstr double`	VT_BOOL, VT_DECIMAL, VT_I1, VT_I2, VT_I4, VT_R4, VT_F8, VT_R8, VT_PTR TO VT_UI4	Integer, Long, Single, Double, Decimal Boolean, Date	sbyte, short, int, uint, ulong, float, double, bool, decimal	System.SByte, System.Int16, System.Int32, System.Int64, System.UInt16, System.UInt32, System.UInt64, System.Decimal
NumVec, IntVec	{F8 I2}, I[], F8[]	`Int[]`	VT_CY, VT_DATE, VT_COLOR, VT_I4[]	Integer(1 to 3)	int[], double[]	
Arr	A	`void[], int[]`	VT_VARIANT, VT_ARRAY	Variant()	object	System.Object, System.Array
ArrEncArr	{I4 T[9]}, {I4 U4 {U1[4]} I4 I4}	`struct`	VT_ARRAY OF VT_VARIANT, VT_SAFEARRAY	Variant(1 to 3, 1 to 4, 1 to 2)	object	System.Object
MatEncArr			VT_VARIANT[;]		int[][,]	System.Object
RefSc			VT_DISPATCH, VT_PTR TO VT_COCLASS	Object		System.Object, etc…
Empty, ⎕*NULL*		`void`	VT_VOID,	Empty, Nothing, Null	void	System.Void
⎕*OR* of space			VT_DISPATCH			

0.3.1 Review **APL1_2.PDF** which introduces mainstream 1st and 2nd generation APLs such as *IBM APL2*, *Dyalog APL*, *STSC APL*PLUS* and *MicroAPL APL.68000*.

Note: You can download **APL1_2.PDF** free from http://www.microapl.co.uk/apl/APL1_2.pdf.