I should say to begin with that this is history from my perspective. If you had a similar presentation from Pete Donnelly or John Daintree or even Morten Kronberg it would be different.

Where we began:

Pauline Brand had a sister in Zilog.

Pauline is a very significant part of Dyalog's history. She was recruited from Atkins when Dyadic Systems was about 4 months old. So 1978. Using Meredith Belbin's team roles she is a classic "finisher".

Zilog had produced one of the really successful 8 bit processors of the late '70s and was looking to produce a 16 bit processor. APL was a significant requirement for computer offerings of the time – thanks to IBM. The 16 bit processor was the Z8000 which was not to see the same success as the Z80. Still the 2 sisters pulled the managements together and an investment was arranged.

Dyadic recruited John Scholes who had left Atkins to go to the European Space Agency and then had implemented APL for ICL. Both for 1900 and 2900 machines.

When it got to the stage of actually implementing I was placed in the team alongside John and managed by David Crossley. John and I were working in Zilog's UK offices in Maidenhead – a riverside town west of London.

This introduced me to a number of new things.

Firstly UNIX which Zilog had chosen as its operating system. It was not until I read an obituary for Dennis Richie that I realised just how early we had got into UNIX.

Secondly screen based development. Until then My experience had been scrolling paper terminals. (Not quite true. At Atkins I had done a statistics/APL project that had used a Tektronix graphics terminal.)

This defined our environment. We had not had much choice so far – it was imposed.

First choice was to use C. This was not obvious. Zilog had its own low level development language which was a serious contender. John Scholes thought C would open up possibilities for a more portable implementation. This proved absolutely true!

Second choice was whether we were to implement a first generation APL or a nested array APL. John Scholes, who had already implemented APL on some restricted hardware (ICL 1900) favoured just doing a first generation APL. I and David Crossley pushed for a second generation APL. This was, probably, our most significant choice but was made almost casually in an afternoon of discussion.

This decision did not affect us for a while. We had code to write that did not depend on it.

I got on with writing the scanner – the bit of code that translates APL text into tokens. John got on with writing the parser which is the bit of code that processes the tokens and calls appropriate primitives. He was also writing the memory management. It should be noted that the scanner was the first bit of C I wrote and

subsequent developers suffer as a result – sorry Silas.

The parser wasn't ready yet so I next coded a large primitive ⎕FMT. Which might seem an odd place to start. Still not very proficient in C – sorry Vince.

The parser was now, sort of, operational so I could code primitives.

At this stage we had to make a choice. Boxed arrays or floating arrays. John's choice was floating – he had been following the discussions.

We had documentation – the NARS blue book. We actually wound up implementing more of it than STSC did. We coded to the only documentation we had so Paul Berry's I P Sharp book, the NARS blue book and Xerox's documentation of their APL implementation.

Since we had nested arrays we changed some of the inputs and outputs of primitives to reflect that.

I was coding two primitives a day on average. Although the set primitives (membership, dyadic ⍳, union, intersection, differ, unique) took longer they used a common approach. I used hash tables for those.

We ran out of code space. Only had 64K. So I developed a way of using multiple processes to get more. Using Unix pipes to pass arguments and results or for some things entire workspaces. Building on these led to the development of auxiliary processors as a means of customers being able to extend the language.

I could now use an innovative approach to )COPY and load one workspace into a process and pipe the appropriate bits back into the user's workspace.

There was a discussion as to whether to implement the user commands )load and the like or use proper primitives. The Dyadic employees who were still using various APLs on a daily basis balked at losing the system commands so we implemented them.

At about this point we had version 1 of Dyalog APL which Phil Goacher could document and use our brown and cream colour scheme for those of you with long memories.

Now we had an APL we could port. The first port was to a Gould which was a 32 bit box – luxury. It also threw up a problem the solution to which was useful later and elsewhere. The problem was that addresses were hard. 32 bit things had to be on 32 bit boundaries. 64 bit things (floating point) had to be on 64 bit boundaries. On a 32 bit machine all of our arrays started on 32 bit boundaries so we had to juggle a bit with doubles. Later the same approach was used with Decimal Floating Point 128 bit alignment and vector processors 256 bit alignment (I think – Marshall did most of the vector processing code.).

We proceeded to port to a lot of new machines. The manufacturers needed to tick the box that they had APL although they did not sell much of it. Talk to Pete about how that worked financially. I just kept flying around the world doing ports. Those days it was easier to move me about than to move machines to me.

We were still UNIX but then came DOS (not the operating system on IBM 360 but the IBM/Microsoft one on Intel 8086). The 80286 was horrible and we avoided it. For a while we sold an add on board that used NatSemi processors and provided UNIX and sensible addressing but it was a high entry cost.

Then came the 80386. It was a game changer. Sensible addressing, PharLap which got DOS into 32 bit and MetaWare Hi C (came packaged with a copy of Mark's gospel) gave us a compiler. MKS Toolkit gave us UNIX like tools to use for our builds. Suddenly (well not quite that quickly) we had a DOS product. Of course we were an application that required multi processing so that we could do those tricks with )COPY and the like. DOS was no help but the 80386 was. It was DOS so we had low level access to everything and the 80386 had task swap segments. So I used those.

Then came Windows 3. Flat address space and I could not use the low level task swap facilities. Still it did have some Multi processing. As a Unix guy I thought it was horribly deficient but we could work with it.

Somewhere about then we did ⎕SM. This is still a good tool for green screen applications but few still do those. Pauline, Pete, and Adam Curtin drove that development and did a fantastic job. I had had some experience with both the Atari ST and the three rivers/ICL Perq. I thought ⎕SM would have a limited life as GUI was coming. I got that both right and wrong. It is still used extensively by one of the largest customers.

Along came John Daintree and he produced ⎕WC. I largely ignored this which was my mistake. The Windows object addressing was buried in a text left argument. However the demand was to bring it out of there and make it part of the language. We had adopted a Windows centric view of the object hierarchy the `big.smaller.smallerstill` whereas I argued strongly for a `smallerstill⊃smaller⊃big` approach. The existing left argument syntax from ⎕WC won that. Still hurts a bit and it made the scanner more complex.

It had another implication that came later, consider

```
big[index1].small[index2]
```

From which namespaces are `index1` and `index2` taken? The way we implemented it was that they are taken from the current namespace. If we had gone the other way then it would be different. I am not saying better. It was some time before John Scholes noticed that we had done this without asking ourselves the question.

⎕WC was interesting in that we had choices to make for the implementation. JD favoured writing direct to the Windows code. This would preclude a UNIX implementation. There was something of a host of possibilities for doing similar on UNIX as well as Windows. I explored Qt a little and there was Mosaic and WxWidgets. None really convinced us that they were a long term investment. JD wrote directly to the Windows API and the resulting ⎕WC was successful. Our UNIX

versions continued to use our green screen ⎕SM. As it has turned out Qt has been successful and long term. So maybe we missed a trick.

At about the same time – early 90s – along came the DEC Alpha. A serious 64 bit chip. I still enthuse about that chip despite it being little endian and wasting silicon doing both VAX and Ieee floating point. It could have done a 128bit floating point instead of the VAX code. We did a port but it always stood alone. We just pushed our types to be 64 bit. There were (still are) issues with not being able to take a 64 bit integer to a 64 bit double and back without a loss of precision. For this port we extended fuzzy comparison to include 64 bit integers.

Development continued: Windows95, Windows NT (I quite liked that).

On a non technical aspect we split the company. IBM insisted that if you sold p-series, about which we knew a lot, you had to sell i-series of which we knew nothing. We split and merged the p-series side with a company that knew i-series. It broke Pauline's heart to do that. We have since dragged Karen and Andy back into the fold but I think we have missed Pauline's skills.

My son started a PhD, evaluating characteristics as emulsions were stirred, and was using APL to throw very large Boolean arrays about. I realised he was going to run out of address space. I bought a second hand Sun UltraSPARC and sent an email at work to say I was going to do a proper 64 bit port in my own time. Pete contacted some customers to see if this was something they wanted and, crucially, would they sponsor it. He got the support and we did the 64 bit port using my UltraSPARC and a newly purchased Intel machine to use the brand new XP64 for Windows. My son dropped out of his PhD following a cycling accident but we had made, what I think was, a crucial development. It was much later that I realised that IBM mainframes were restricted to a 2GiB address space and thus IBM's APL2 had that restriction. Being a Linux/Unix guy I should point out that the restriction comes from Z/OS (should that be pronounced zee OS?) not the hardware.

JD got into .Net in a big way but it didn't affect me very much. It probably would if I was not retired as it has now reached Linux but not AIX.

John Scholes did his skunkworks Dfns project. Along with the Dfns workspace this was a tour-du-force and a real boost for APL. The only issue, from my point of view, is the static referencing. I preferred programming with dynamic referencing. The static referencing has benefits with mathematical code analysis. Is anybody actually doing that with APL code?

The next development was a learning curve for me as I did ⎕MAP. The learning was about paging and memory layout. I could map a single page with our own pocket header to precede the map of a file. The learning I did for this was crucial for the work on Shared Code Files – see my presentation a few years ago.

Roger Hui came and brought with him "trains". Yet another style of coding and loved by some. I was now old and decrepit and, worse, did not write much APL code so didn't catch the bug. I think my younger self would have done so.

The last decision was taken when I gave a years notice that I planned to retire at age 75. I had planned to cut to a 4 day week at 70, 3 day week at 71, and so on. This failed as I realised that I might be able to cut my time but the "crap" did not reduce. So I stuck at 3 days a week. When I said I was to retire at 75 it was decided that I would spend a year doing internal presentations on the code. Ask the guys – especially Silas if this has proved useful. It certainly caused me to do a lot of revision as I worked through code that was, in some cases, decades old. JD, John Scholes and I went to Microsoft in Seatle during the .Net development. I was sent because “Geoff thinks differently”. I made it the main purpose of the code handover to enable, the poor souls, who had to inherit the code to be able to understand something of how different that thinking had been.